UNIVERSIDAD POLITÉCNICA DE CATALUÑA

TRABAJO FINAL DE GRADO

Evaluación de algoritmos de multiplicación de matrices dispersas portados a un entorno GPGPU

Autor:
Joan MARCUAL MEDINA

Supervisores:
Beatriz Otero
Agustín Fernández

19 de enero de 2017

Índice

1 Introducción

Actualmente la mayoría de estudios científicos son llevados a cabo haciendo uso de simulaciones. Las simulaciones presentan ventajas como: la opción de repetir los experimentos todas las veces se quiera, poder cambiar parámetros físicos y la más importante resultan más económicas que la experimentación real.

Figura 1: Simulación de una mandíbula

Para realizar estas simulaciones se debe discretizar el problema haciendo uso de una malla de puntos que simule el objeto de estudio. En la figura 1 podemos ver un ejemplo de la simulación de una mandíbula [1]. Con las relaciones de adyacencia impuestas en cada punto, se relaciona el valor de un conjunto de variables incógnitas definidas en cada nodo y denominadas grados de libertad [2]. Este conjunto de relaciones se puede representar como un sistema de ecuaciones lineales. El número de ecuaciones de dicho sistema es proporcional al número de nodos.

Para resolver los sistemas de ecuaciones se usan matrices. Estas matrices resultan ser de un tamaño enorme, algunas superando los cientos de millones de filas [5], debido a que tienen tantas filas como puntos tiene el mallado. Además, la gran mayoría de los elementos de estas matrices son ceros. Estas matrices de gran tamaño donde la mayoría de sus elementos son nulos son conocidas como matrices dispersas.

La operación producto matriz dispersa vector (SpMV) es la operación fun-

damental en este cálculo. Ejecutándose de forma iterativa hasta llegar a una solución. Como resultado, las simulaciones tienen su rendimiento altamente ligado al rendimiento de esta operación. El problema de esta operación es que la gran mayoría de las operaciones son inútiles, ya que casi todos los elementos son ceros (99 % elementos nulos).

La operación básica del producto SpMV se muestra en la figura 2. En ella solo hay 2 operaciones aritméticas por cada 3 accesos a memoria. Además, estos accesos debidos a la gran dispersad de las matrices son accesos irregulares. La combinación de muchos accesos y accesos irregulares hace que la operación este limitada por ancho de banda a memoria.

$$C_i += A_{ij} * B_j$$

Figura 2: Operación matriz por vector

Una solución al problema de la gran cantidad de cálculos inútiles es la compactación de las matrices en formatos donde no se guarden los ceros. Además, algunos de ellos intentan regularizar los accesos a memoria de la matriz.

Durante los últimos años se ha popularizado el uso de las tarjetas gráficas como unidad de cálculo científico. Las tarjetas gráficas poseen una arquitectura paralela que encaja a la perfección en muchos problemas científicos con un gran coste computacional o de ancho de banda a memoria. Su arquitectura está especialmente diseñada para ejecutar problemas con un alto grado de paralelismo. En las aplicaciones de cálculo científico es habitual encontrar este tipo de paralelismo.

La utilización de tarjetas gráficas se ha mostrado como una solución para mejorar el rendimiento de la operación SpMV. El gran ancho de banda de las GPUs encaja a la perfección con las necesidades de este problema.

En este proyecto se analizan los rendimientos de varios formatos de representación de matrices dispersas portados a un entorno GPU.

2 Estado del arte

La optimización de la operación SpMV ha sido un tema recurrente de estudio en los últimos años. Esto es debido a que muchas aplicaciones científicas contienen esta operación como parte crucial en su cálculo. El rendimiento de esta operación es muy pobre (habitualmente, sobre el 10 %) en CPUs [7]. Es por esta razón, que se han propuesto muchas mejoras para intentar incrementar este rendimiento. Aunque todas estas mejoras estén orientadas a paliar el efecto cuello de botella que genera el acceso a memoria, estas pueden ser clasificadas según su enfoque. El primer enfoque es el pre-procesado de la matriz, el segundo la compresión de datos y el tercero la propuesta de nuevos formatos de representación de matrices dispersas.

2.1 Pre-procesado

Las soluciones propuestas de pre-procesado de matrices están en su mayor parte basadas en el reordenamiento. Este enfoque trata de conseguir mejores patrones de distribución de los elementos en la matriz. El algoritmo de referencia en este campo es el de Cuthill-McKee [8]. Este algoritmo consiste en permutar la matriz para maximizar el número de elementos no nulos en la diagonal principal. De esta forma, haciendo uso del formato DIA [9], se consigue una gran reducción del espació necesario para guardar la matriz.

Otra solución parecida a la anterior es la propuesta por R. Amestoy et al. que propone un algoritmo de reducción del grado de la matriz [10]. Como resultado de aplicar este algoritmo se consigue una matriz con una distribución del número de elementos por columna más uniforme. Por ultimo también se debe hacer mención al algoritmo de función distancia presentado por Pichel, Singh y Carretero [11]. El objetivo de esta técnica es aumentar los grupos de elementos no nulos de la matriz, lo que permite con esta nueva matriz explotar la localidad espacial en memoria. Esta última técnica, combinada con blocking y tiling, es la que mejor rendimiento ha conseguido hasta el momento.

2.2 Compresión de datos

La operación producto matriz dispersa vector esta limitada por la transmisión de datos entre memoria y la unidad de procesamiento. Por ello, la compresión de datos es una respuesta lógica a esta problemática. WillCock y Lumsdaine

[12] fueron los primeros en proponer un método usando compresión sin pérdidas para reducir el volumen de transmisión de datos. Su propuesta estaba basada en Delta encoding y run-length encoding. De forma parecida, Kourtis et al. [13] usó compresión de datos aplicada a los índices y a los valores consiguiendo una aceleración de hasta 1.2x.

Aunque estas dos versiones dieron buenos resultados, ambas estaban pensadas para el uso en CPU. Estas soluciones para CPU no pueden ser adaptadas al cálculo de SpMV en GPU como consecuencia de las grandes diferencias que hay entre sus arquitecturas. Es por esto, que Teng et al. [14] presentaron una nueva propuesta consciente de las arquitectura para GPUs. En ella se abandonan las técnicas de compresión basadas en entropía y se usa una compresión a nivel de representación de bits. Con ella llegaron a conseguir una media de aceleración de 1.5x.

2.3 Formatos de representación de matrices dispersas

En general, las matrices se almacenan en memoria en posiciones consecutivas de memoria. Este formato es perfecto para matrices densas, pero no de adapta a las necesidades de una matriz dispersa. Esto es debido al gran número de elementos nulos que son guardados. Por esta razón se han presentado nuevos formatos que se adaptan mejor a este tipo de matrices. Un resumen y análisis de los formatos básicos es presentado en [9].

A partir de estos formatos básicos se han presentado variaciones para mejorar algunos de sus aspectos. Por ejemplo Barbieri et al. propusieron una variación del formato ELL (ELL-G) [15], donde se guardaba la matriz de forma column-major. De esta forma, se explota el coalescing a nivel de Warp. Otra mejora del formato ELL (ELL-R) es propuesta en [16] donde se usa un nuevo vector para almacenar el número de elementos de cada fila. Mejorando el formato ELL-R se propuso el formato ELL-Based [17], que propone una forma de reducir el tamaño del nuevo vector.

3 Mi proyecto

Mi proyecto presenta una solución al problema de la operación de SpMV y su evaluación de rendimiento. El trabajo propone la implementación de la operación utilizando diferentes formatos de almacenamiento, haciendo uso de programación GPGPU.

Mi proyecto contará con la implementación de ocho formatos existentes de almacenamiento de matrices dispersas, y sus respectivos algoritmos de multiplicación. Para cada formato habrá dos implementaciones, implementación en serie y implementación en GPU. Para realizar las implementaciones de GPU se usará CUDA [3].

Los formatos de almacenamiento implementados serán: COO, CSR, DIA, ELL, ELL-Based, ELL-G, HYB, HYB-G.

Una vez implementaos todos los formatos, se hará un análisis exhaustivo de rendimiento comparando los diferentes tiempos de ejecución en un conjunto de matrices seleccionadas para la prueba.

3.1 Objetivos

Los principales objetivos de este proyecto son:

- La implementación de 8 algoritmos de multiplicación de matrices dispersas en un entorno GPU basados en los formatos COO, CSR, DIA, ELL, ELL-Based, ELL-G, HYB, HYB-G.
- La evaluación del rendimiento en operaciones de punto flotante por segundo (FLOPS) de estos 8 algoritmos.
- La evaluación de rendimiento de los diferentes formatos en diferentes tipos específicos de matriz.

3.2 Actores implicados

A continuación se identifican las partes interesadas en este proyecto. Los actores implicados, a quién va dirigido y quién se beneficiara de sus resultados.

- **Desarrollador de proyecto**: Tiene interés en conseguir un buen resultado final que mejore el rendimiento de la operación y cumpla todos los requisitos.

- **Responsables del proyecto**: Agustín Fernández, en especial Beatriz Otero, tiene interés en que el proyecto tenga un buen resultado ya que investiga en este campo.
- **NVIDIA**: Tiene interés en el buen resultado de este proyecto. Ya que esto podría derivar en una mejora de la actual librería de matrices dispersas de NVIDIA.
- **Colectivo científico**: Cualquier científico involucrado en simulaciones se verá beneficiado por el resultado favorable de este proyecto.

4 Metodología

4.1 Ejecución

Este trabajo sera ejecutado sobre el servidor BOADA del Departamento de Arquitectura de Computadores la Facultad de Informática de Barcelona. Este servidor esta preparado para poder ser accesible mediante Internet. De esta forma, la mayoría de este trabajo podrá ser realizado remotamente des de mi ordenador personal. Los pasos para poder ejecutar el proyecto en el servidor son los siguientes.

1. Enviar el código al servidor.
2. Esperar en cola hasta que el servidor esté disponible.
3. Esperar a que termine la ejecución del código.
4. Recoger los resultados.
5. Comprobar los resultados

4.2 Validación

La validación es uno de los aspectos más importantes en la Informática. Para validar que los algoritmos SpMV implementados sean correctos se ejecutarán un gran número de matrices dispersas. Estas matrices serán extraídas de la colección de matrices dispersas de la Universidad de Florida [5]. Una vez ejecutados se hará una comprobación automatizada mediante un script para validar que los resultados sean correctos.

4.3 Control de versiones y seguimiento

Para realizar un control de versiones del código se ha elegido la herramienta GIT [6]. Es la herramienta más estandarizada para este propósito. Además, se hará uso del cliente GitHub para una mayor facilidad si se produce algún problema.

Los supervisores de este proyecto tendrán acceso total a estos repositorios pudiendo ver el estado del proyecto en cualquier parte de su desarrollo. Además, se concertarán citas periódicas cada dos semanas para mostrar el estado del trabajo.

5 Planificación

En este apartado se explican las diferentes tareas que se han realizado durante el proyecto, desglosadas por bloques y ordenadas cronológicamente según el diagrama de Gantt que se adjunta al final de la sección.

5.1 Duración del proyecto

La duración del proyecto ha sido de 4 meses. Se empezó a principios de Setiembre y se esperaba acabar la segunda semana de Diciembre. Esta fecha de entrega se retraso, por motivos que se explican más adelante, a la primera semana de Enero.

5.2 Descripción del proyecto

En la mayor parte del proyecto se sigue una metodología en cascada. De esta manera las tareas de desarrollo se ordenan de forma secuencial, vistas hacia abajo como una cascada). Las tareas de este trabajo están divididas en 4 bloques. Siendo estos: Las tareas de GEP, la preparación del entorno de trabajo, el desarrollo de los algoritmos y la redacción del documento.

A continuación se describen los diferentes bloques.

GEP: En este bloque se presentan las fechas de entrega de las diferentes tareas de la asignatura de Gestión de Proyectos. Estas entregas son útiles para la planificación del proyecto. Debido a esto, son realizadas en una parte muy temprana del mismo. En el diagrama de Gantt podemos ver que las primeras entregas se solapan con el bloque de desarrollo de los algoritmos. Estos dos bloques se retro-alimentan entre si, siendo el estudio de los algoritmos imprescindible para las primeras entregas y siendo estas entregas útiles para la organización del desarrollo.

Estas son las diferentes entregas y sus fechas de vencimiento:
Entregable 1: Definición del alcance y connaturalización. 27/09/2016
Entregable 2: Planificación temporal. 03/10/2016
Entregable 3: Gestión económica y sostenibilidad. 10/10/2016
Entregable 4: Presentación preliminar. 17/10/2016
Entregable 5: Documento de Especialidad. 24/10/2016

Entregable 6: Documento único. 24/10/2016

Preparación del entorno: Este bloque es el primero en orden cronológico ya que ninguno de los siguientes puede ser llevado a cabo sin la finalización de este. El bloque consta de dos tareas: la preparación del entorno en casa y en el servidor. Se usaron dos semanas para ello. Una para la instalación de Linux en mi pc personal, junto con los drivers de NVIDIA. Otra para la creación de una cuenta en el servidor y la configuración del entorno en esta cuenta.

Desarrollo de los algoritmos: Este es el bloque que forma el núcleo del proyecto. Esta divido según el esquema clásico de desarrollo de software. En esta metodología todas las tareas son dependientes de las anteriores. Donde se empieza con un estudio de la solución. Le sigue la implementación de la solución. Después se pasa por la fase de validación. Por último en el caso especifico de este proyecto se realizarán unos tests de rendimiento.

Documento: Este bloque solo consta de la redacción del documento final del trabajo de final de grado. Esta debe ser dejada para la ultima tarea a realizar ya que es totalmente necesario haber acabado las anteriores satisfactoriamente.

5.3 Riesgos

5.3.1 Caída del servidor

Este proyecto es realizado en un servidor remoto por lo que no se tiene un control total sobre él. Esto conlleva una serie de riegos: el primero es que la cola este sobre-utilizada, lo que conllevaría a grandes esperas de tiempo cada vez que se quisiera ejecutar el código pudiendo llegar a ralentizar el desarrollo normal del proyecto. El segundo y mayor riesgo de todos es el de la fallida total de este servidor.

En el primero de los casos se podría llegar a un acuerdo con el departamento para conseguir más prioridad en la cola de ejecución. En el segundo, si el servidor se estropeara se realizarían las evaluaciones de rendimiento en mi ordenador personal a una escala mucha más pequeña.

5.3.2 Validación errónea

La validación errónea es un problema inherente en todo desarrollo de software. Para evitarla se usará una gran muestra de entradas intentando contemplar

todos los casos posibles. También se usará una herramienta automática para facilitar este trabajo. Si aún así se encontrara algún error en la implementación de los algoritmos durante cualquier etapa del proyecto, se le daría máxima prioridad a la corrección de este error.

5.4 Estimación de las horas

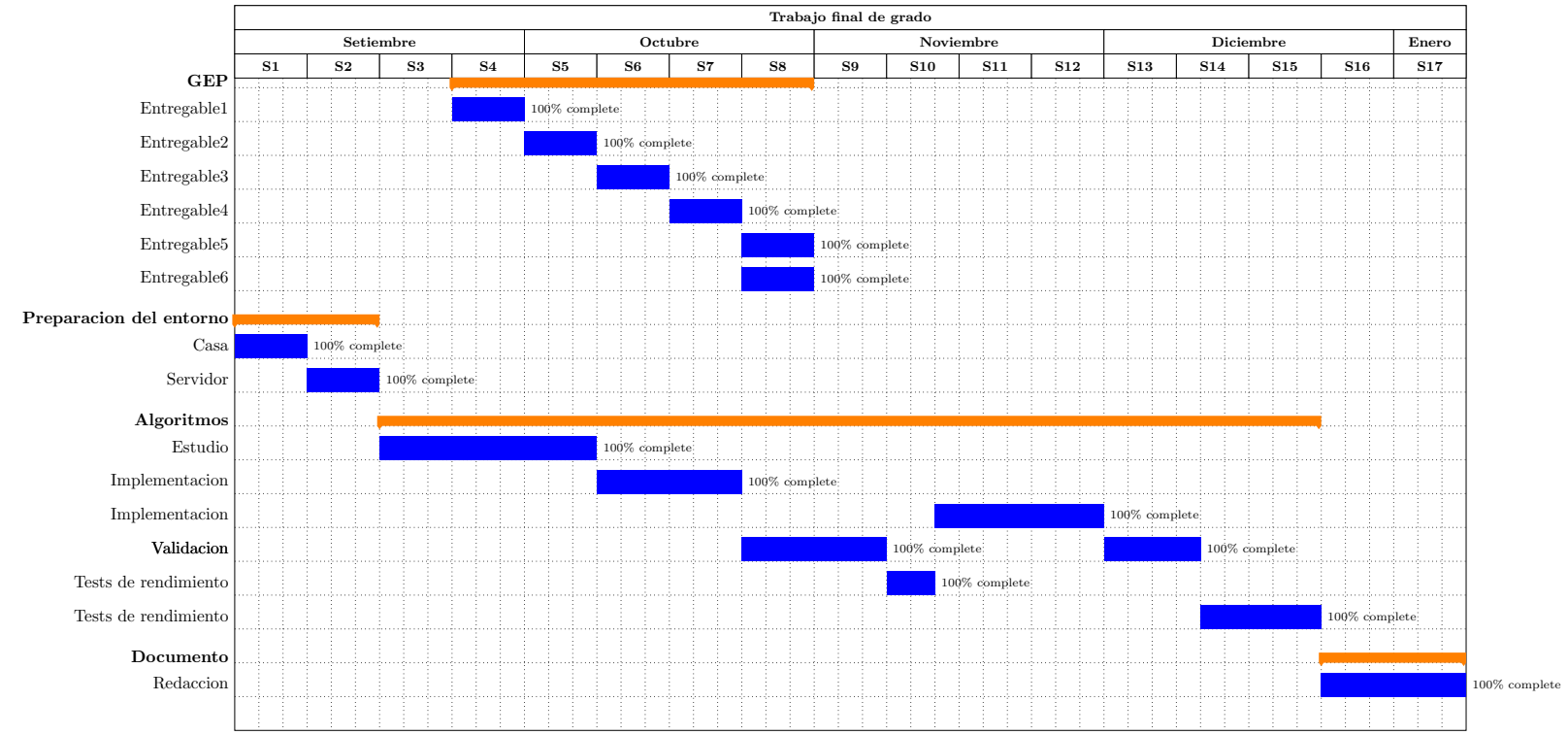
GEP	108		**Preparación del entorno**	18
Entregable 1	24		Casa	10
Entregable 2	10		Servidor	8
Entregable 3	11			
Entregable 4	8		**Desarrollo de Algoritmo**	230
Entregable 5	25		Estudio	60
Entregable 6	30		Implementación	60
			Validación	40
Documento	80		Test de rebdimiento	70
Redacción	80			
			Total	436

5.5 Desviaciones

En la fase de test de rendimiento se encontraron problemas en los algoritmos al ejecutar contra el juego de pruebas seleccionado. Este imprevisto ya estaba contemplado, por ello la fecha prevista de finalización estaba estimada con margen de prórroga. Como contingencia se retomó la fase de implementación para solventar estos problemas. Después de esta fase se tubo que volver a validar la solución y una vez validada se pudo pasar otra vez a la fase de análisis de rendimiento. En el diagrama de Gantt ya esta plasmado este cambio.

5.6 Diagrama de Gantt

Figura 3: Diagrama de Gantt



6 Identificación y estimación de costes

Para el cálculo de costes de este proyecto se debe tener en cuenta que se trata de un proyecto de final de carrera sin beca asociada al mismo. Es por este motivo que se ha intentado minimizar los costes necesarios para el desarrollo de este proyecto. Para conseguir esto, se usará solo software libre y se trabajará aprovechando equipamiento hardware ya existente.

6.1 Costes recursos humanos

Para el desarrollo de este proyecto habrá dos personas responsables y un desarrollador. Para calcular las horas de trabajo de cada integrante se ha seguido el siguiente criterio. Para el desarrollador se han sumado las horas de las diferentes tareas vistas en el diagrama de Gantt, ya que estas corresponden a todas las actividades que debera realizar durante el proyecto. Para los dos responsables se han contado las horas de reunión semanales que tendrán con el desarrollador.

Rol	**Salario**	**Horas de dedicación**	**Coste final**
Desarrollador	15 euros/hora	460 horas	6.900 euros
Responsable	20 euros/hora	60 horas	1.200 euros
Responsable	20 euros/hora	60 horas	1.200 euros
Total			**9.300 euros**

6.2 Costes hardware

Los equipos involucrados para el desarrollo de este proyecto son dos. Un portátil donde el desarrollador programara la solución y el servidor donde la se ejecutará. El portátil es el ordenador personal del desarrollador y el servidor es el servidor Boada del Departamento de Arquitectura de Computadores de la FIB. La amortización está calculada basándose en la vida útil de un equipo de 4 años y el porcentaje de dedicación del equipo en este proyecto.

Equipamiento	**Coste**	**Amortización al año**	**Tiempo**	**% de uso**	**Coste estimado**
Portátil	615€	25 %	4 meses	%100	51.25 euros
Servidor	20.000€	25 %	4 meses	%5	83.3 euros
Total					**134.55 euros**

6.3 Costes software

Como se ha indicado anteriormente, se trabajará solo con programas de licencia gratuita de manera que el coste total serán 0 euros.

Nombre del software	**Coste de la licencia**
Ubuntu 14.04	0 euros
CUDA	0 euros
GIT	0 euros
Skype	0 euros
Sublime Text	0 euros

6.4 Costes Total

El coste total es la suma del coste de recursos humanos, costes de hardware y costes de software.

	Coste
Recursos humanos	9.300€
Hardware	134.55€
Software	0€
Total	**9.435€**

7 Sostenibilidad

7.1 Sostenibilidad económica

Este proyecto no tiene como objetivo conseguir beneficios económicos. Aun así, cabe la posibilidad de conseguirlos. Si los resultados de rendimiento del proyecto son muy favorables, esta tecnología podría ser vendida a NVDIA para añadirla a su librería de multiplicación de matrices dispersas.

7.2 Sostenibilidad social

El objetivo de este proyecto es la optimización de una operación clave en muchas investigaciones, ejemplos de ello son: simulaciones sísmicas, investigaciones biomédicas como cálculos de posicionamiento de proteínas, investigaciones de mallados eléctricos, entre muchas otras. Esta posible reducción del tiempo de la operación SpMV reduciría el tiempo de las simulaciones de estas investigaciones. Además, con la reducción del coste que esta operación implica se podrían usar modelos mucho más complexos y por tanto más reales de los que se usan actualmente.

7.3 Sostenibilidad ambiental

Para la elaboración de este proyecto se han gastado recursos materiales, que han consumido recursos en su construcción y hacen uso de energía eléctrica. Aun así, estos recursos son usados durante toda su vida útil. El ordenador personal donde se programa el proyecto se seguirá usando una vez la terminación de este. El servidor es usado paralelamente por otros desarrolladores y seguirá siendo usado una vez de finalice este proyecto.

Si los resultados del proyecto son exitosos, la aplicación de mismo supondría un ahorro energético en las aplicaciones que hagan uso de operación SpMV.

7.4 Matriz de Sostenibilidad

La matriz de sostenibilidad está basada en "la economía del bien común", de Christian Felber [20]. Donde un mayor número total implica un proyecto más sostenible, siendo el máximo 90 y el mínimo -60.

El análisis de la sostenibilidad del proyecto se divide en tres partes, identificadas por las columnas de la matriz.

- El proyecto puesto en producción (PPP), que incluye la planificación, el desarrollo y la implantación del proyecto.
- La vida útil del proyecto, que empieza una vez implantado y acaba en su desmantelamiento.
- Los riesgos, inherentes del propio proyecto durante toda su construcción y su vida útil.

	PPP	**Vida útil**	**Riesgos**
Ambiental	7	15	0
Económica	6	5	-3
Social	7	1	0
rango de sostenibilidad	20	21	-3
Total		**38**	

8 Contexto

8.1 GPGPU

GPGPU o *General-Purpose Computing on Graphics Processing Units* es un concepto en Informática en el que se usan tarjetas gráficas (GPUs) para el cálculo de aplicaciones muy costosas en cómputo.

Las tarjetas gráficas en los últimos años han conseguido superar drásticamente la potencia y ancho de banda a las CPUs (figura 4). La ventaja más importante de las GPUs contra las CPUs es su precio. Si comparamos el precio del rendimiento entre ellas ($/Flop) obtenemos que en CPUs el precio esta en 11$/GFLOP [21] y en cambio en GPUs está en 0.08$/GFLOP [22].

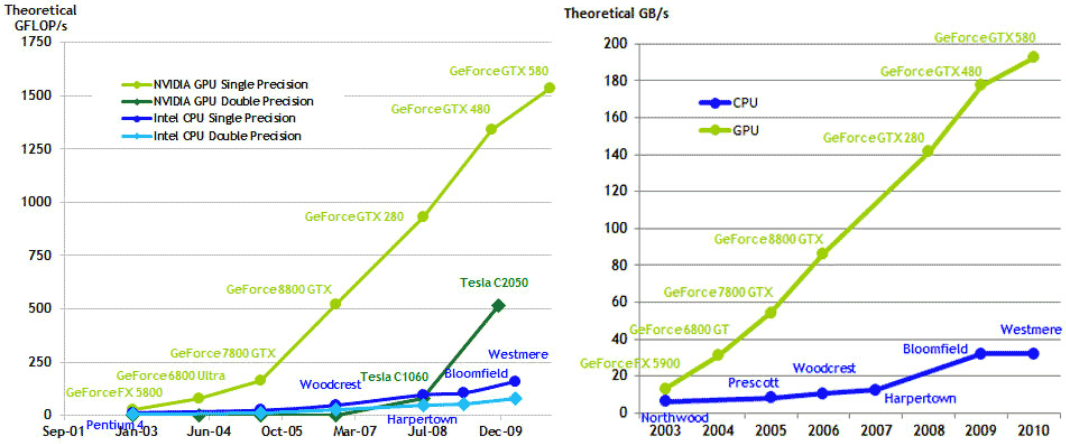


Figura 4: rendimiento GPU vs CPU

El motivo por el cual no se usan siempre GPUs es por sus diferencia de arquitectura con las CPUs (figura 5). Las CPUs están típicamente compuestas de varios núcleos, cada uno de ellos con mucha potencia de cálculo. Por contra, las GPUs están compuestas por muchísimos núcleos cada uno de ellos con una potencia de cálculo relativamente pequeña. Esta característica hace que dependiendo del problema sea mejor optar por una CPU o una GPU.

Se utiliza GPGPU en problemas altamente paralelizables, donde se puede puede aplicar una distribución del trabajo equitativa. El trabajo es distribuido en la gran la cantidad de núcleos que poseen las GPUs.

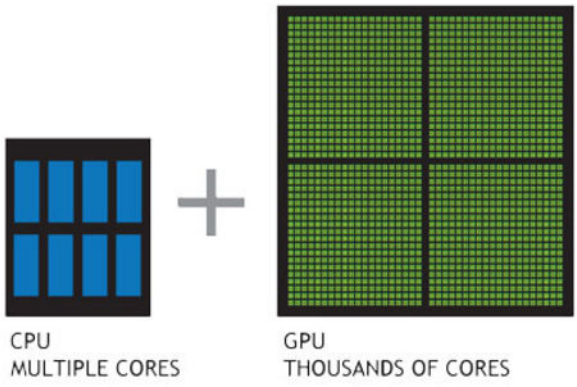


Figura 5: Arquitectura GPU vs CPU

8.1.1 Arquitectura GPU

Las GPUs tienen una gran cantidad de núcleos, estos núcleos están organizados en bloques (figura 6). Cada bloque puede acceder a la memoria de la GPU y además tiene su propia memoria caché.

Las tareas son divididas en bloques y la unidad de control de ejecución va asignando tareas a bloques de threads. Cuando un bloque de threads es liberado se le asigna una nueva tarea y así hasta que no queden tareas. Un bloque de threads es liberado cuando todos sus threads han terminado sus tareas. Por ello, si el trabajo no está equitativamente distribuido entre los threads del bloque, el thread más lento marcará el tiempo gastado en esa tarea.

Cada núcleo no posee una unidad de proceso independiente, sino que es compartida con todo el bloque. Esto es, cada bloque contiene una unidad de proceso vectorial. En esta unidad de proceso vectorial se puede ejecutar la misma instrucción para todos los threads a la vez. Esta característica hace que para conseguir el mejor rendimiento todos los threads deben ejecutar la misma instrucción simultáneamente. Por ello, códigos con diferentes ramas de ejecución en el mismo bloque de threads penalizarán el rendimiento de la ejecución.

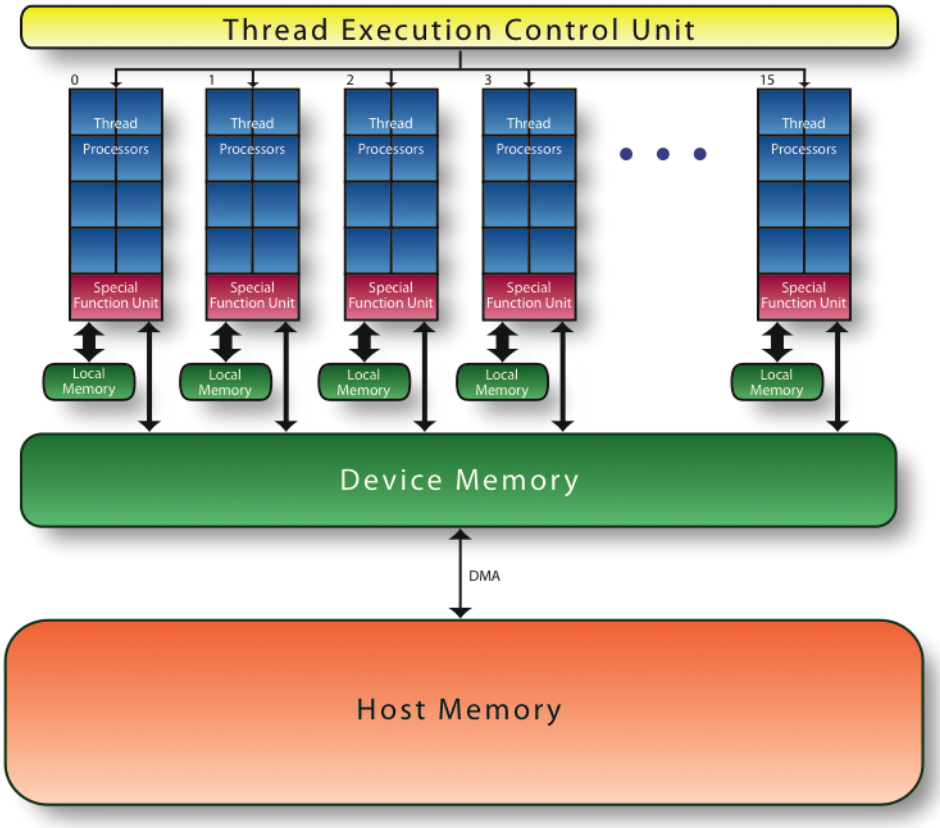


Figura 6: Arquitectura GPU

8.1.2 CUDA

CUDA [3] es una plataforma diseñada por NVIDIA de forma conjunta a nivel de software y hardware para hacer uso de la potencia de cálculo de las GPUs en aplicaciones de uso general. Con ella, tenemos la opción de usar directivas en códigos C, C++ y Fortran para que algunas partes del código sean ejecutadas en la GPU. Además, también nos brinda la opción de hacer uso de la memoria de la tarjeta gráfica. CUDA es la plataforma GPGPU más usada. A pesar de todo, esta tecnología no es multiplataforma. CUDA solo puede ser usado en tarjetas gráficas de la marca NVIDIA.

En la figura 7 vemos un ejemplo de código CUDA para realizar la operación

SAXPY (Single-Precision A·X Plus Y). En la invocación del ejemplo se pasan dos parámetros, el primero (4096) indica el número de bloques y el segundo (256) indica el número de threads por bloque. En esta rutina CUDA cada thread hace una sola operación SAXPY. Para saber que posición del vector debe operar cada thread se calcula la posición mediante la siguiente formula: el ID del bloque por la dimensión de cada bloque más el ID del thread en ese bloque.

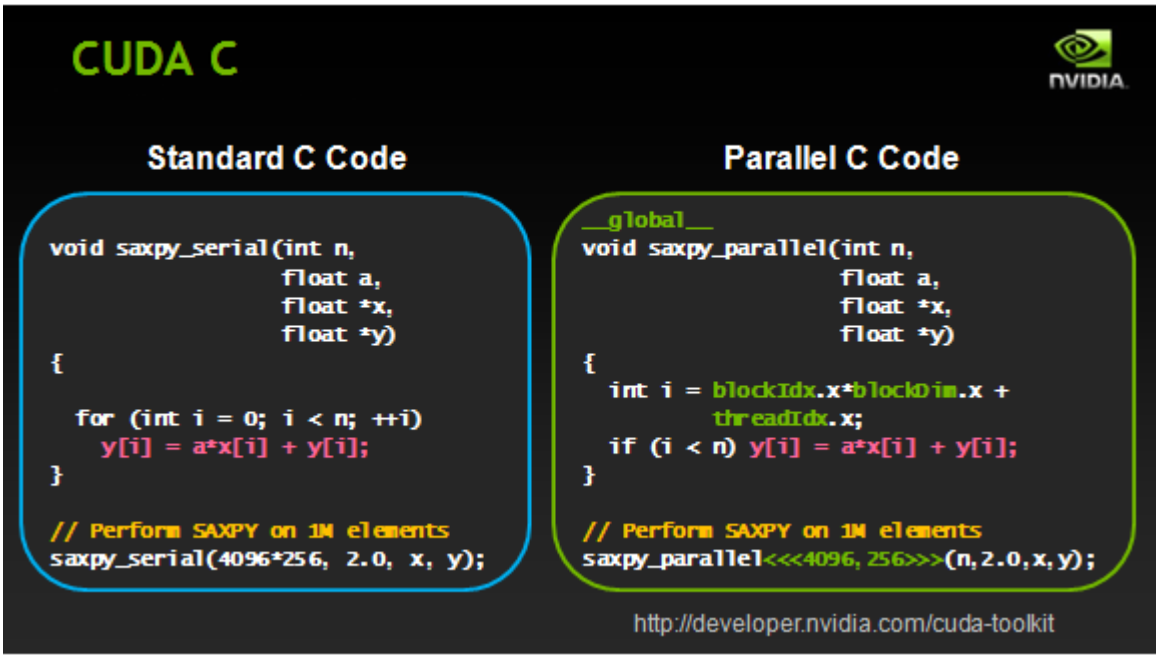


Figura 7: Ejemplo código CUDA

8.2 Multiplicación matriz dispersa vector

Una matriz dispersa es una matriz de gran tamaño donde la mayoría de los elementos que la componen son nulos. En la figura 8 podemos ver una matriz que ejemplifica un matriz dispersa a una escala reducida. El elemento c_i del vector producto C se obtiene multiplicando cada elemento de la fila i de la matriz A por cada elemento del vector B y sumándolos (figura 9). El problema que representa esta operación en matrices dispersas es que la mayoría de los cálculos no son provechosos, puesto que la mayoría de los elementos son cero.

$$A_{m,n} = \begin{pmatrix} 2 & 0 & 5 & 0 & 0 & 8 \\ 0 & 15 & 0 & 0 & 4 & 0 \\ 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Figura 8: Ejemplo reducido de matriz dispersa

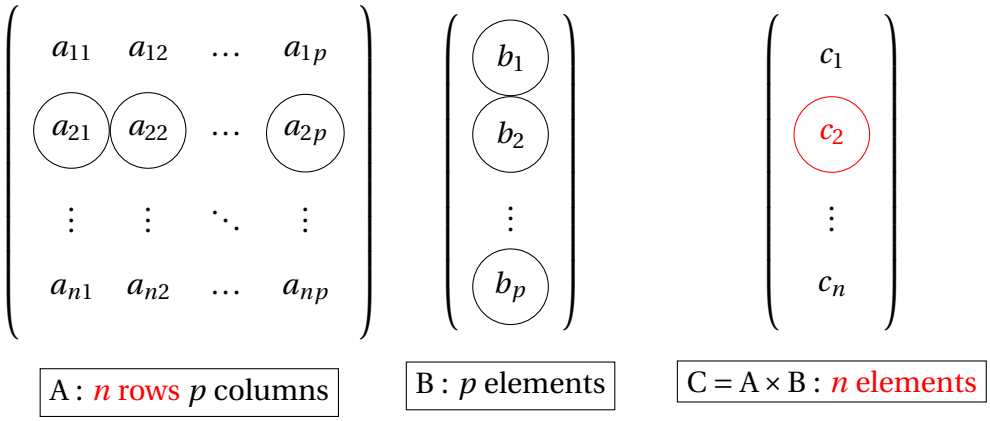


Figura 9: Algoritmo estándar de multiplicación matriz vector

8.3 Formatos de representación de matrices dispersas

Usando la matriz de la figura 8, a continuación explicaremos los formatos de almacenamiento a implementar en este proyecto.

8.3.1 COO

El formato de coordenadas (COO) [9](figura 10) es el sistema más básico de representación de matrices dispersas. Este usa tres arrays para representar la matriz. El primer array es usado para representar los valores no nulos de la matriz. Otro es usado para posicionar en que fila de la matriz esta cada elemento no nulo. Por ultimo se usa otro array para posicionar en que columna se halla el elemento.

8.3.2 CSR

El formato Compressed Sparse Row (CSR) [9](figura 11) es una mejora sobre el formato COO. Uno de los mayores problemas del COO es el malgasto de me-

$$Valores_{nz} = \begin{pmatrix} 2 & 5 & 8 & 15 & 4 & 7 & 1 \end{pmatrix}$$
$$Filas_{nz} = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 2 & 3 \end{pmatrix}$$
$$Columnas_{nz} = \begin{pmatrix} 0 & 2 & 5 & 1 & 4 & 2 & 3 \end{pmatrix}$$

Figura 10: Ejemplo formato COO

moria al guardar las columnas y las filas. Para reducir este gasto el formato CSR propone reducir el array de filas.

Cogiendo como ejemplo la (figura 10) podemos ver que en el array de filas hay los mismos valores de fila repetidos de forma secuencial. Esto ocurre típicamente debido a que se recorre la matriz de fila en fila. Estas repeticiones pueden ser substituidas por un único número que represente el número de elementos guardados hasta la fila anterior. De esta forma el array Filas apunta a la posición del primer elemento de la fila a tratar en el array Columnas. Esto reduce tanto el número de memoria necesaria para almacenar la matriz como el número de accesos en la operación.

$$Valores_{nz} = \begin{pmatrix} 2 & 5 & 8 & 15 & 4 & 7 & 1 \end{pmatrix}$$
$$Filas_{nz} = \begin{pmatrix} 0 & 3 & 5 & 6 \end{pmatrix}$$
$$Columnas_{nz} = \begin{pmatrix} 0 & 2 & 5 & 1 & 4 & 2 & 3 \end{pmatrix}$$

Figura 11: Ejemplo formato COO

8.3.3 DIA

El formato Diagonal (DIA) [9](figura 12) esta fuertemente pensado para matrices donde los elementos no-nulos están concentrados cerca de la diagonal principal de la matriz. En este formato la matriz dispersa es representada por una matriz donde cada fila es una diagonal de la matriz dispersa. Además se usa un array donde se indica a que diagonal pertenece cada fila. Siendo la diagonal principal la 0, las inferiores el número negativo de la distancia a la diagonal principal y las superiores el número positivo de la distancia a la diagonal principal.

Este formato al ser especializado tiene un rendimiento altamente ligado a

la estructura de la matriz. Es por ello que solo es recomendable para matrices banda.

$$\mathrm{V}alores = \begin{pmatrix} 2 & 15 & 7 & 1 \\ 5 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 8 & 0 & 0 & 0 \end{pmatrix}$$

$$\mathrm{I}ndices = \begin{pmatrix} 0 & 2 & 3 & 4 \end{pmatrix}$$

Figura 12: Ejemplo formato DIA

8.3.4 ELL

En el formato ELL [18](figura 13) la matriz dispersa es representada mediante el uso de dos matrices densas. Estas matrices tienen el mismo número de filas que la original pero menos columnas.El número de columnas K es el máximo de elementos no nulos que se pueden hallar en una fila en la matriz dispersa.

La primera matriz contiene los elementos no nulos de la matriz original. La segunda matriz contiene el número de la columna donde está situado el elemento en la matriz original. Adicionalmente, las dos matrices contienen elementos de padding cuando el número de elementos por fila es inferior a K.

$$\mathrm{V}alores = \begin{pmatrix} 2 & 5 & 8 \\ 15 & 4 & * \\ 7 & * & * \\ 1 & * & * \end{pmatrix}$$

$$\mathrm{I}ndices = \begin{pmatrix} 0 & 2 & 5 \\ 1 & 4 & * \\ 2 & * & * \\ 3 & * & * \end{pmatrix}$$

Figura 13: Ejemplo formato ELL

8.3.5 ELL-Based

El formato ELL-Based [17](figura 14) es una mejora del formato ELL. El problema que tiene el formato ELL es el padding que se debe añadir a todas las filas hasta que tengan K elementos. Siendo K el número máximo de elementos no-nulos encontrados en una fila de la matriz. Esto conlleva un malgasto de memoria en matrices donde el número de elementos por fila no es uniforme.

En el formato ELL-Based se aplica ELL sobre bloques de filas en vez de sobre toda la matriz. Por cada bloque se calcula la K máxima y se guarda en un vector. De esta forma conseguimos mitigar el coste en memoria cuando la matriz tiene mucha variabilidad en el número de elementos no-nulos por fila.

$$\mathrm{T}ama\tilde{\mathrm{n}}\mathrm{o}\mathrm{B}loque = 2filas$$

$$\mathrm{K} = \begin{pmatrix} 3 & 1 \end{pmatrix}$$

$$\mathrm{V}alores_{nz} = \begin{pmatrix} 2 & 5 & 8 \\ 15 & 4 & * \\ 7 & & \\ 1 & & \end{pmatrix}$$

$$\mathrm{I}ndices_{nz} = \begin{pmatrix} 0 & 2 & 5 \\ 1 & 4 & * \\ 2 & & \\ 3 & & \end{pmatrix}$$

Figura 14: Ejemplo formato ELL-Based

8.3.6 ELL-G

El formato ELL-G [16](figura 15) es una variación del formato ELL. El formato ELL-G se diferencia del ELL en que los elementos son guardados de forma Row Major a diferencia del ELL que son guardados de manera Column Major. La colocación de los elementos en memoria condiciona en gran medida el rendimiento de los accesos. Siendo crucial la elección de un formato o el otro dependiendo de la arquitectura donde la operación SpMV se ejecutará.

$$\mathrm{V}alores = \begin{pmatrix} 2 & 15 & 7 & 1 \\ 5 & 4 & * & * \\ 8 & * & * & * \end{pmatrix}$$

$$\mathrm{I}ndices = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 2 & 4 & * & * \\ 5 & * & * & * \end{pmatrix}$$

Figura 15: Ejemplo formato ELL-G

8.3.7 HYB

En el formato híbrido [19](HYB) se combinan los formatos ELL y COO. En este formato se usa el formato ELL para la parte regular y el COO para la parte irregular. En este formato K no es fijada como el número máximo de elementos de una fila, sino que se usa una heurística para decidir el mejor valor de K según la distribución del número de elementos no nulos entre las filas. De esta forma en matrices muy irregulares en número de elementos por fila se mitiga el sobre coste que suponen los elementos de padding. Para el ejemplo de la figura 16 fijamos K en 2.

$$\mathrm{V}alores\mathrm{ELL} = \begin{pmatrix} 2 & 5 \\ 15 & 4 \\ 7 & * \\ 1 & * \end{pmatrix}$$

$$\mathrm{I}ndices\mathrm{ELL} = \begin{pmatrix} 0 & 2 \\ 1 & 4 \\ 2 & * \\ 3 & * \end{pmatrix}$$

$$\mathrm{V}alores\mathrm{COO} = \begin{pmatrix} 8 \end{pmatrix}$$

$$\mathrm{F}ilas\mathrm{COO} = \begin{pmatrix} 0 \end{pmatrix}$$

$$\mathrm{C}olumnas\mathrm{COO} = \begin{pmatrix} 5 \end{pmatrix}$$

Figura 16: Ejemplo formato HYB

8.3.8 HYB-G

El formato HYB-G es una evolución del formato HYB donde se usa el formato ELL-G en vez del formato ELL.

$$\mathrm{V}alores\mathrm{ELLR} = \begin{pmatrix} 2 & 15 & 7 & 1 \\ 5 & 4 & * & * \end{pmatrix}$$

$$\mathrm{I}ndices\mathrm{ELLR} = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 2 & 4 & * & * \end{pmatrix}$$

$$\mathrm{V}alores\mathrm{COO} = \begin{pmatrix} 8 \end{pmatrix}$$

$$\mathrm{F}ilas\mathrm{COO} = \begin{pmatrix} 0 \end{pmatrix}$$

$$\mathrm{C}olumnas\mathrm{COO} = \begin{pmatrix} 5 \end{pmatrix}$$

Figura 17: Ejemplo formato HYB-G

9 Pruebas

Se han utilizado dos ejecutables para las pruebas:

- **Test SpMV Serie**: realiza 1000 iteraciones sobre la operación SpMV para los siguientes formatos: COO, CSR, ELL, ELL-Based, ELL-G, HYB, HYB-G, DIA.
- **Test SpMV CUDA**: realiza 1000 iteraciones sobre la operación SpMV para los siguientes formatos portados a la plataforma CUDA: COO, CSR, ELL, ELL-Based, ELL-G, HYB, HYB-G, DIA.

El tamaño ideal de bloque de threads varía según el formato y la matriz al que se aplica, para poder comparar entre ejecuciones los más iguales posibles se ha decidido usar el mismo tamaño de bloque para todas las pruebas. El tamaño de bloque de threads usado ha sido 1024. En el formato ELL-Based se ha usado como tamaño de bloque el tamaño del bloque de threads. En el formato HYB se ha usado como heurística de K la media de elementos no-nulos por fila.

Las pruebas se han realizado sobre el siguiente conjunto de matrices extraído de la colección de matrices de la Universidad de Florida [5], en el anexo hay incluida más información sobre estas matrices.

Nombre	***Filas***	**Columnas**	**NNZ**	**%NNZ**
OPF_3754	15435	15435	141478	0.05 %
e40r0100	17281	17281	553562	0.18 %
memplus	17758	17758	126150	0.04 %
msc10848	10848	10848	1229778	1.04 %
Na5	5832	5832	305630	0.90 %
nmos3	18588	18588	386594	0.11 %
psmigr_1	3140	3140	543162	5.50 %
raefsky4	19779	19779	1316789	0.34 %
tandem_vtx	18454	18454	253350	0.07 %
sme3Da	12504	12504	874887	0.56 %
t3dl	20360	20360	509866	0.12 %

10 Resultados

10.1 Serie

En la figura 18 podemos ver los resultados de la ejecución del test SpMV serie. Donde cada barra indica el rendimiento (más es mejor) de la ejecución del formato en cada matriz.

En ejecución serie el formato con mejor resultado es el formato CSR. Seguido por el formato HYB y COO. Los formatos con peor resultado en serie son el formato DIA y el Formato ELL-G. A continuación se analizan los resultados de cada formato en mayor profundidad.

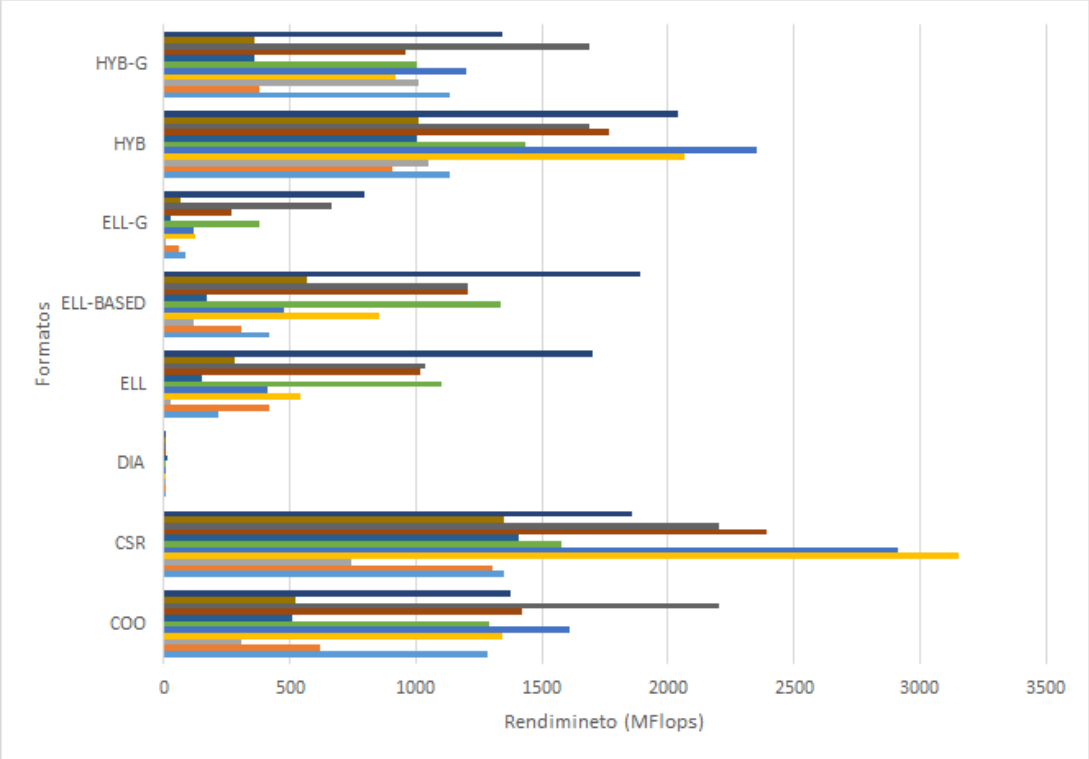


Figura 18: Test SpMV serie

10.1.1 Formato COO

EL formato COO es el formato más simple de representación de matrices dispersas. Es un formato general por lo que el tipo de matriz no debería afectar a su rendimiento. Aun así hay diferencias de rendimientos en las diferentes matrices (figura 19). Esto es a causa de los accesos al vector B en la multiplicación.

La matriz donde consigue mayor rendimiento (tandem_vtx) contiene patrones de bloques. De esta forma los accesos al vector B son por lo general bastante próximos consiguiendo buena localidad espacial en caché.

Por contra, la matriz donde consigue el peor rendimiento (memplus) contiene un patrón de diagonales. Para el acceso por filas del formato COO se traduce en accesos muy salteados en el vector B, conllevando un peor redimiendo en caché.

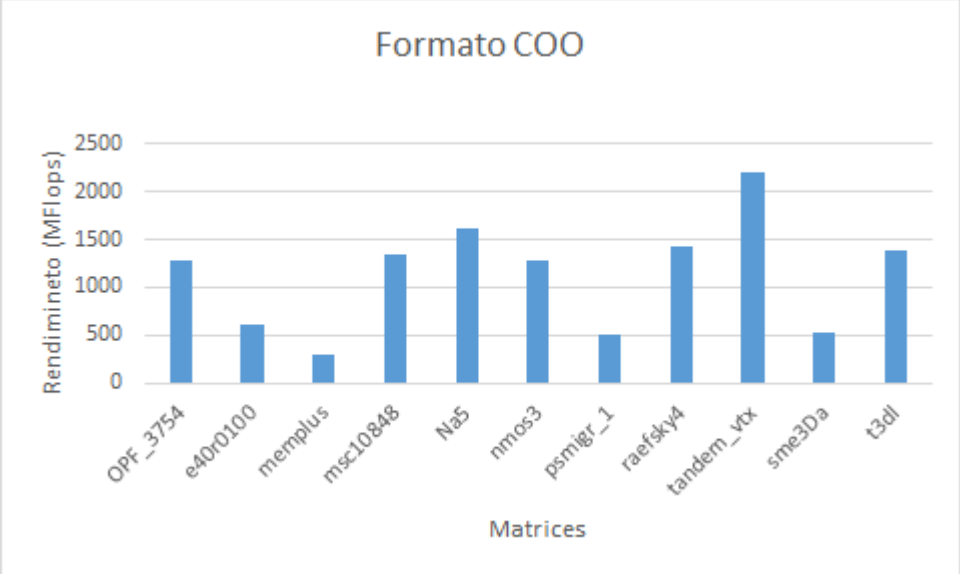


Figura 19: Test COO serie

10.1.2 Formato CSR

El formato CSR siendo la evolución del formato COO consigue unos resultados similares pero mejores en todos los casos (figura 20). Puesto que el formato CSR solo hace un acceso al array de filas por fila, es más premiado por matrices muy densas en número de elementos por fila. Un caso donde se puede ver esta

mejora es en la matriz msc10848, siendo esta muy densa en filas, se consigue una mejora de 2 veces el rendimiento contra el formato COO.

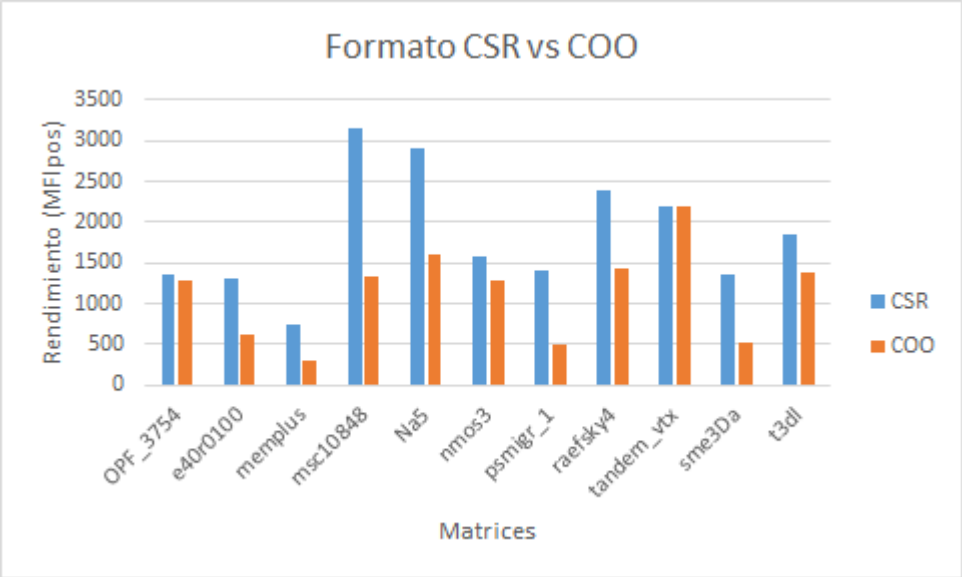


Figura 20: Test CSR serie

10.1.3 Formato DIA

El formato DIA es un formato especializado en matrices en bandas. Por ello, su rendimiento esta altamente ligado a los patrones de la matriz. En la figura 21 podemos ver las grandes diferencias de rendimiento entre matrices. El mejor rendimiento lo consigue en la matriz psmigr_1. Esta es una matriz relativamente densa en diagonales, por lo que no malgasta un gran número de elementos al coger una diagonal no vacía.

Este formato consigue un rendimiento pobre en comparación a los demás (figura 18). Esto es causa de que es un formato muy especifico y en esta prueba se ejecuta contra todo tipo de matrices. A pesar de esto cuando se ejecuta contra matrices en banda reales tampoco consigue un buen rendimiento.

Para mejorar el rendimiento se debería considerar la opción de obtener una solución aproximada de la operación. Una posible mejora sería eliminar los puntos aislados de la matriz, dado que estos causan un deterioro importante en el rendimiento. Cada punto obliga a guardar toda la diagonal donde se encuentra.

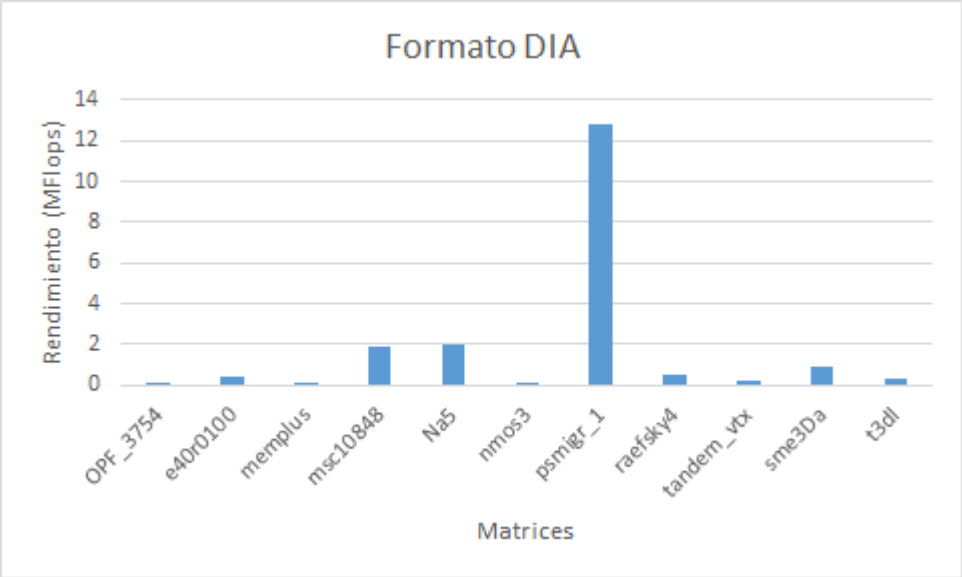


Figura 21: Test DIA serie

10.1.4 Formato ELL

El formato ELL compacta la matriz dispersa en dos matrices densas. Estas matrices densas tiene como número de columnas el número máximo de elementos no-nulos que hay en una misma fila de la matriz dispersa.

Debido a esto, el rendimiento del formato ELL esta ligado a la uniformidad de la distribución de número de elementos no-nulos por fila.

En la matriz donde mejor rendimiento consigue es la matriz t3dl (figura 22) que tiene un patrón de diagonales, que genera un número uniforme de elementos no-nulos por fila.

Por contra el peor rendimiento lo consigue la matriz memplus, esta matriz contiene unas filas iniciales muy densas en comparación a las demás. Esto fija el número de columnas de las matrices ELL a un número más grande de lo necesario en la mayor parte de la matriz. Como resultado a esto hay un gran número de operaciones de elementos nulos.

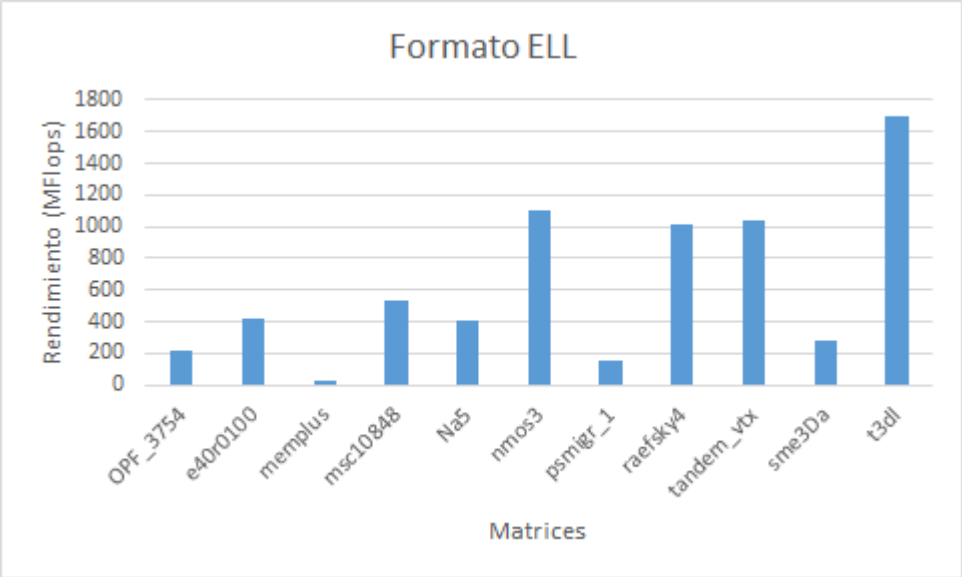


Figura 22: Test ELL serie

10.1.5 Formato ELL-Based

El formato ELL-Based es una evolución del formato ELL. Es por ello que consigue un mejor rendimiento en la mayoría de los casos. Este formato mitiga el malgasto de recursos cuando hay una distribución de elementos por fila no uniforme.

Podemos ver en la figura 23 como en la matriz memplus, la peor del conjunto para el ELL, como el formato ELL-Based consigue mejorar hasta tres veces el rendimiento contra el ELL.

Aun así, no siempre es mejor el formato ELL-Based, en matrices donde el tamaño del bloque elegido no consiga reducir el malgasto de memoria, la inclusión de otro acceso a un vector (vector de tamaño de filas) crea una penalización que puede conllevar a un peor rendimiento que el formato original. Estas condiciones se dan en la matriz e40r0100.

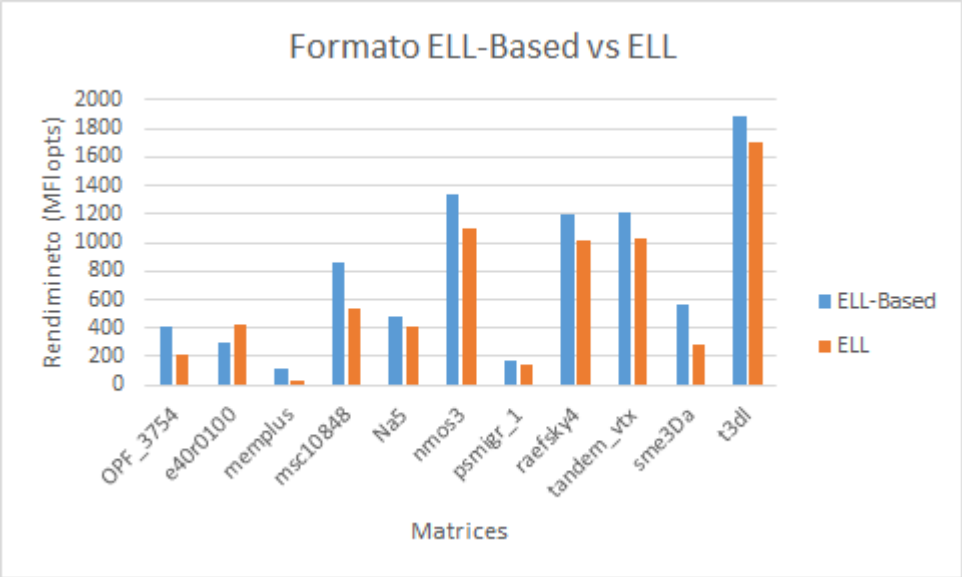


Figura 23: Test ELL-Based serie

10.1.6 Formato ELL-G

El formato ELL-G es una evolución del formato ELL. Este formato esta diseñado para mejorar los accesos a memoria en la plataforma CUDA. Este cambio no solo no mejora la ejecución serie sino que la empeora (figura 24). A causa de la forma de almacenar la matriz se pierde la localidad espacial en la ejecución en serie. Este cambio produce una penalización en tiempo de acceso a memoria que hace bajar el rendimiento total de la ejecución.

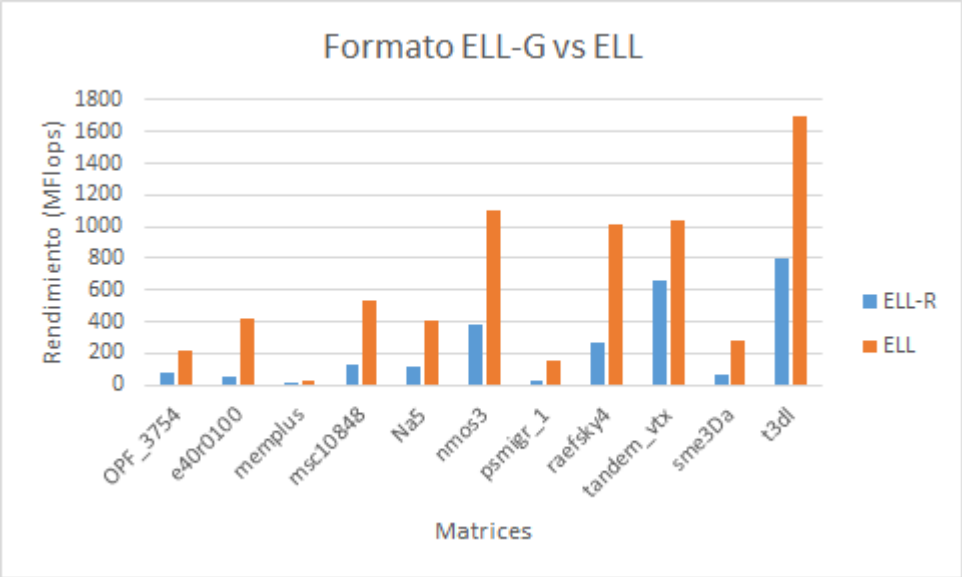


Figura 24: Test ELL-G serie

10.1.7 Formato HYB

El formato HYB es una combinación del formato ELL y el formato COO. Usando como base del formato el formato ELL y usando el formato COO para las filas con más elementos que la media. En la gráfica de la figura 25 podemos ver una comparativa del formato ELL contra sus dos componentes. Este formato, aunque esta pensado para la plataforma CUDA, consigue un gran rendimiento en ejecucción serie.

El hacer uso de una parte COO para filas muy densas atenúa el problema raíz del ELL. En matrices como memplus, msc1084g y sme3Da el rendimiento llega ha ser más del doble que el mejor de los dos casos por separado.

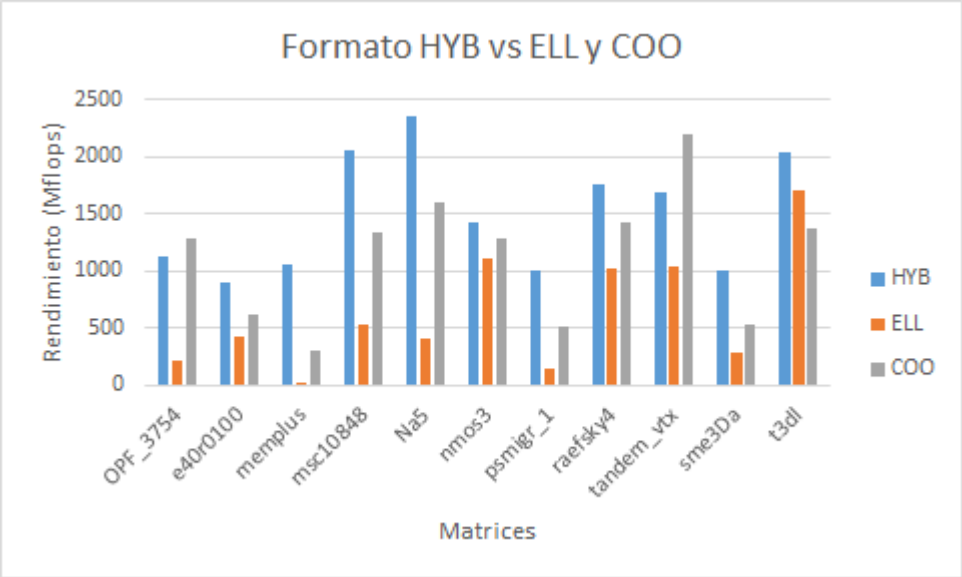


Figura 25: Test HYB serie

10.1.8 Formato HYB-G

El formato HYB-G es una evolución del formato HYB. Esta evolución esta pensada para la plataforma CUDA. Es por ello que en una ejecución serie el rendimiento no se ve mejorado sino empeorado debido al cambio de patrón de acceso a memoria. En la figura 26 podemos ver como consigue un rendimiento peor que el formato original.

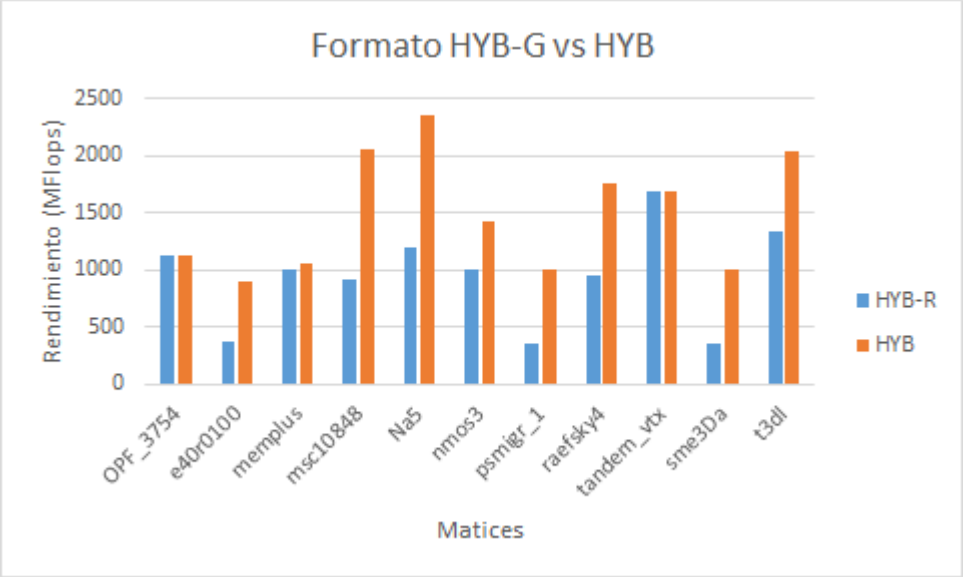


Figura 26: Test HYB-G serie

10.2 CUDA

En la figura 27 podemos ver los resultados de la ejecución del test SpMV CUDA. Donde cada barra indica el rendimiento de la ejecución del formato en una matriz. En CUDA el formato con mejor resultado es el formato HYB-G. Seguido por el formato ELL-G y HYB. Los formatos con peor resultado en CUDA son el formato DIA y el Formato ELL. A continuación se analizan los resultados de cada formato en mayor profundidad

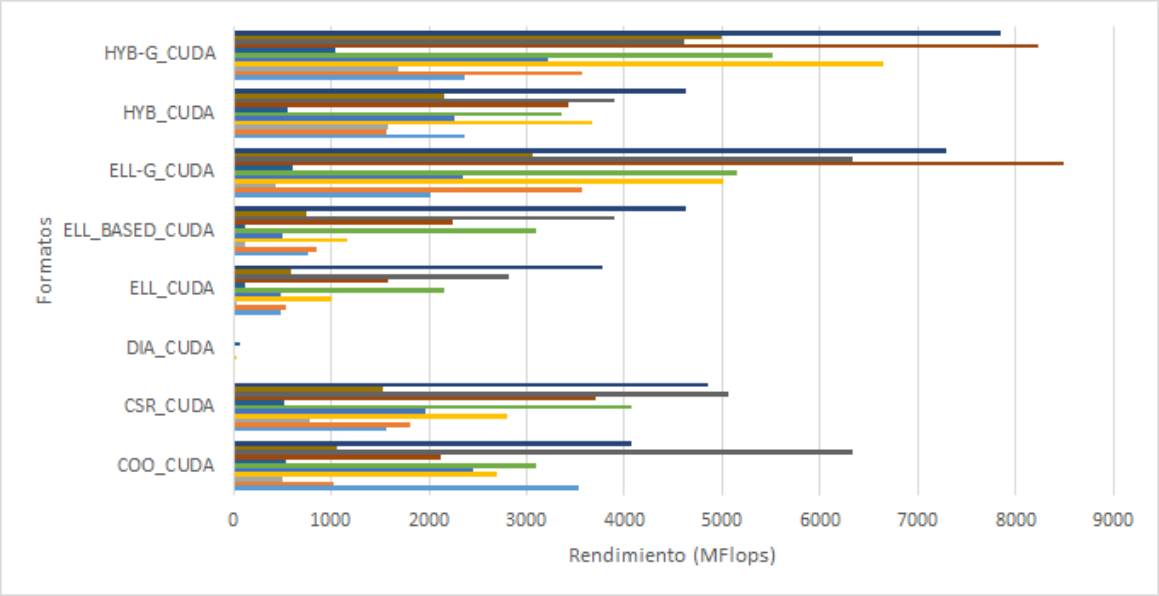


Figura 27: Test SpMV CUDA

10.2.1 Formato COO CUDA

El formato COO no es especialmente idoneo para la plataforma CUDA. Debido a su naturaleza no podemos repartir la matriz la por filas. A causa de esto, varios threads pueden hacer accesos simultáneos de resultados parciales en el vector resultado. Para evitar condiciones de carrera se debe de hacer uso de instrucciones atómicas en los accesos a este vector.

En la figura 28 vemos que ha habido una mejora en todos los casos, pero en algunos debido a que tienen más colisiones de acceso al vector resultado que otros la mejora ha sido menor.

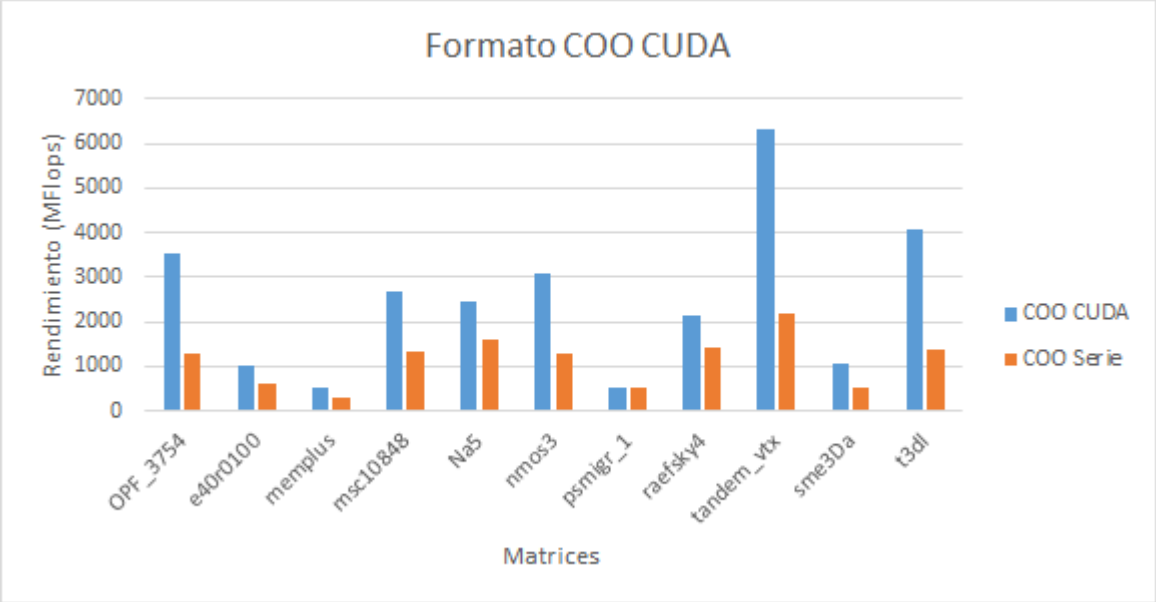


Figura 28: Test COO CUDA

10.2.2 Formato CSR CUDA

El formato CSR se adapta bastante bien a la plataforma CUDA. En este formato si que podemos repartir la matriz por filas, evitando así problemas de consistencia entre threads. Esto da más flexibilidad al formato.

En la figura 29 vemos una mejora en la mayoría de matrices. Aun así hay algunas matrices donde COO tiene más rendimiento que CSR. Este fenómeno es resultado de la espera de threads de cada bloque. Cada bloque de threads de

CUDA debe esperar a que todos los threads del mismo acaben la ejecución. Es por ello que aquellas filas con mayor número de elementos no nulos que sus adyacentes penalizan a todo el bloque.

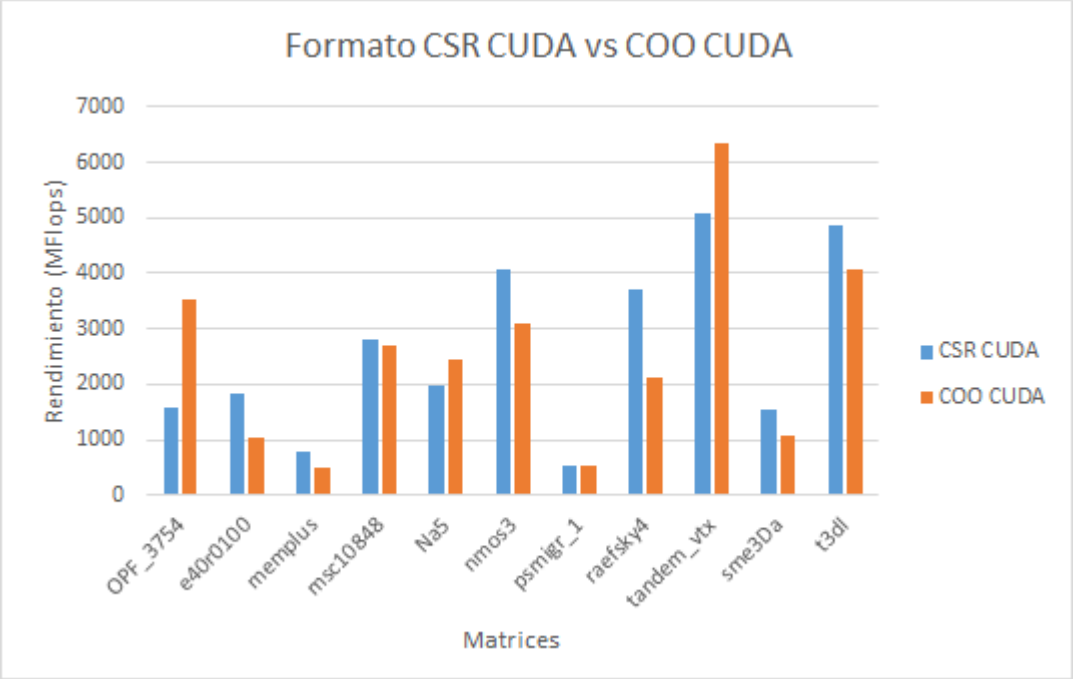


Figura 29: Test CSR CUDA

10.2.3 Formato DIA CUDA

El formato DIA no es un formato pensado especialmente para la plataforma CUDA. Por ello la división del trabajo no es idónea. En la figura 30 vemos una mejora de rendimiento en todas las matrices pero siguen teniendo un rendimiento muy por debajo de los demás formatos.

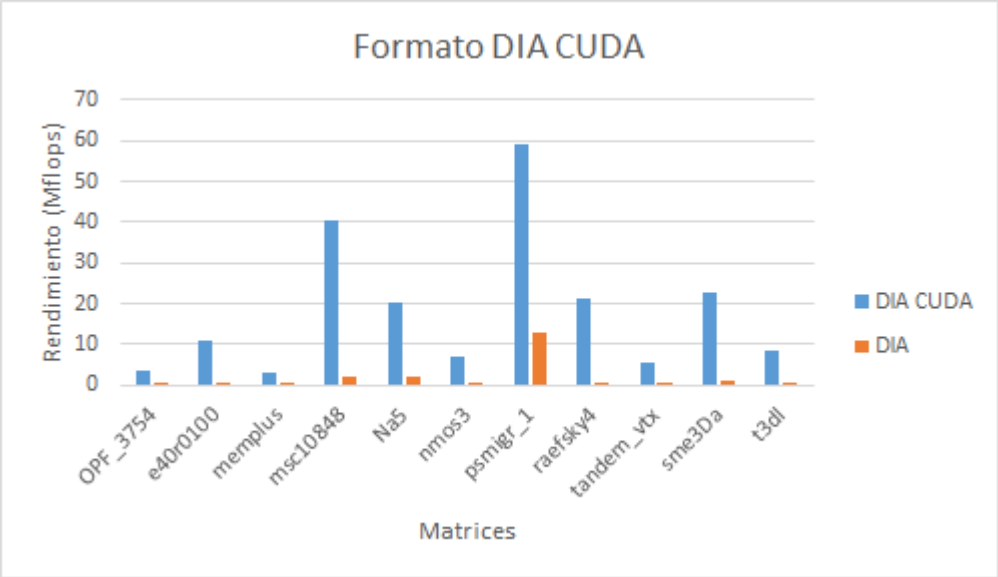


Figura 30: Test DIA CUDA

10.2.4 Formato ELL CUDA

El formato ELL es un formato idóneo para la plataforma CUDA en cuanto a repartición de trabajo. La repartición del trabajo es totalmente equitativa entre los diferentes threads por lo que nunca se da una penalización por un thread lento.

En la figura 31 vemos una mejora en todas las matrices respecto a la versión serie. Aunque todavía persiste el problema de la uniformidad de número de elementos por fila.

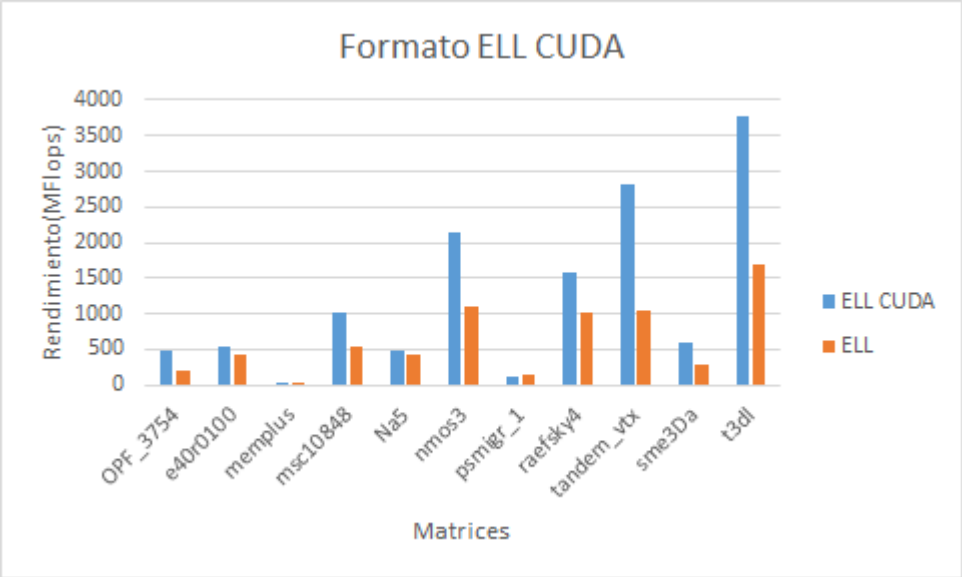


Figura 31: Test ELL CUDA

10.2.5 Formato ELL-Based CUDA

El formato ELL-Based se adapta perfectamente a la arquitectura de CUDA. La opción más natural es la de usar un tamaño de bloque múltiple al tamaño del bloque de threads. Si no se usa un múltiple obligaríamos a que dos bloques de diferente tamaño convivieran en el mismo bloque de threads. Esto retrasaría al bloque con tamaño más pequeño conllevando un sobre coste en la ejecución.

Como se ve en la figura 32 el formato ELL-Based se ha adaptado bien a CUDA consiguiendo un mejor rendimiento que el ELL en todos los casos.

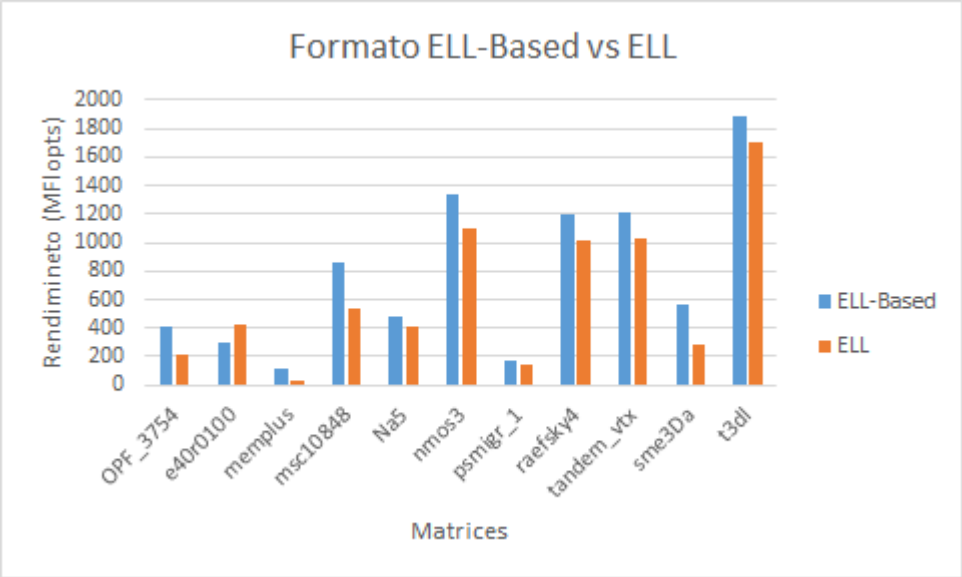


Figura 32: Test ELL-Based CUDA

10.2.6 Formato ELL-G CUDA

El formato ELL-G es una mejora del formato ELL pensada para mejorar el patrón de acceso a memoria en la plataforma CUDA. En la figura 33 podemos ver una mejora importante respecto al formato ELL. Además se postula como uno de los mejores formatos para la plataforma CUDA (figura 27).

En CUDA el patrón de acceso a memoria, con una repartición de trabajo por filas, es de todos los elementos de una columna a la vez. Por tanto, si guardamos la matriz de forma natural en C (row major) cada thread accede a una posición de la caché distinta. En cambio, en el formato ELL-G se guardan los elementos de forma column major. Esto consigue reducir las peticiones a memoria haciendo que toda la fila de caché sea usada simultáneamente.

Además, como ya pasaba en el formato ELL todos los threads tienen una carga de trabajo equitativa. Evitando sobre costes de espera de threads.

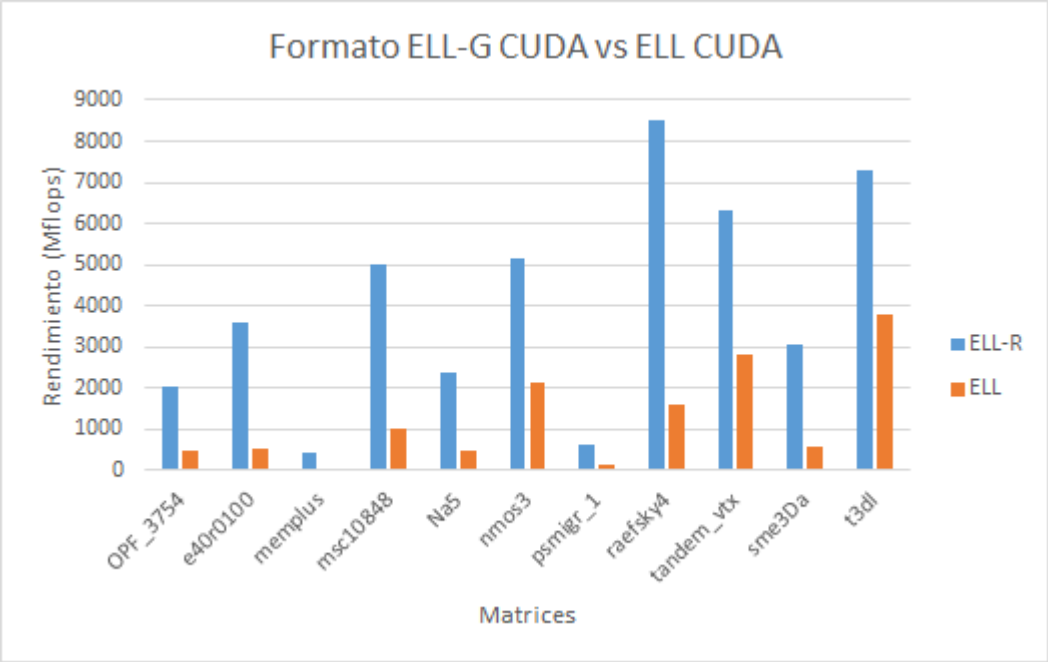


Figura 33: Test ELL-G CUDA

10.2.7 Formato HYB CUDA

El formato HYB en la plataforma CUDA consigue una mejora en la mayoría de los casos (figura 34). En algunas matrices hay un rendimiento peor que el del COO. Esto es a causa de que el formato HYB hace uso del ELL como formato base y COO como apoyo. En aquellas matrices donde COO era claramente mejor que ELL COO sigue teniendo mejor rendimiento que el formato HYB.

El formato HYB en CUDA se presenta como un formato muy útil por su flexibilidad. No consiguiendo el mejor rendimiento en todos los casos pero manteniendo siempre un rendimiento en un rango aceptable.

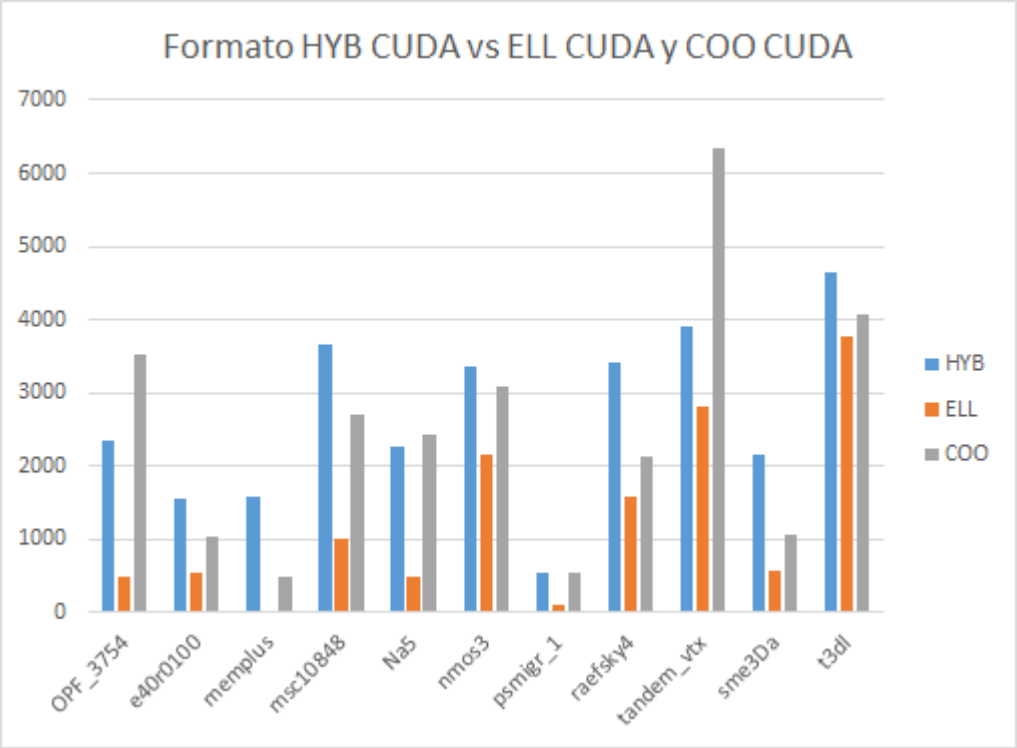


Figura 34: Test HYB CUDA

10.2.8 Formato HYB-G CUDA

El formato HYB-G es el formato con mejor rendimiento en CUDA (figura 27). Si lo comparamos con el formato HYB (figura 33) vemos que consigue mejorar en la mayoría de casos, en los que no lo consigue iguala el rendimiento.

Los casos donde el rendimiento se mantiene son aquellos donde el formato ELL daba un bajo rendimiento, y era la parte COO la que lo mejoraba. Por tanto, una mejora en la parte ELL no tiene impacto en el rendimiento total de estos casos.

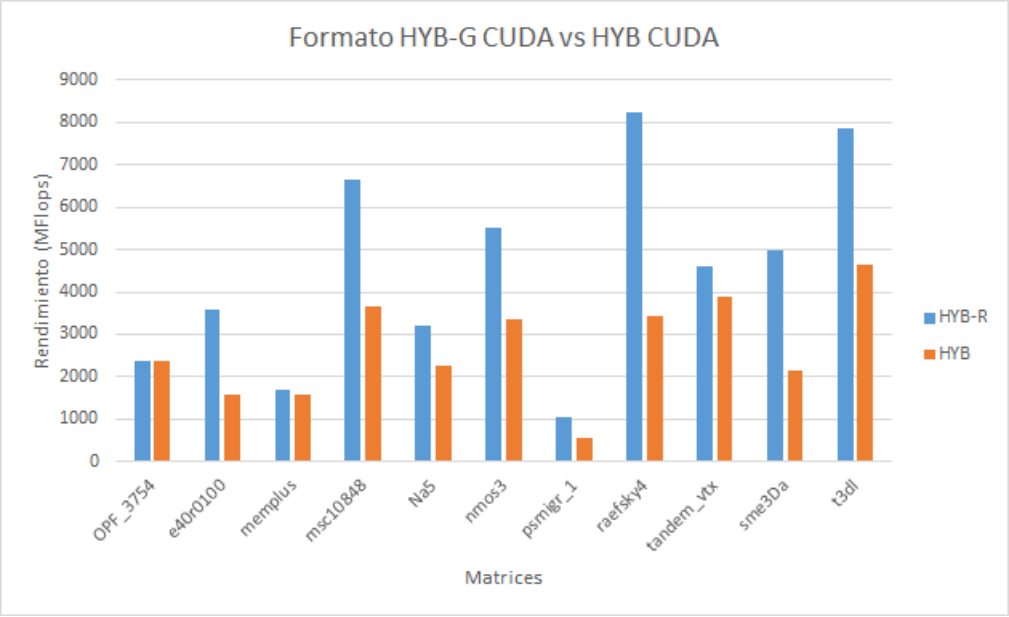


Figura 35: Test HYB-G CUDA vs HYB CUDA

11 Speedup

En la gráfica de la figura 36 vemos una comparativa del speedup contra el tamaño de la matriz. Podemos ver como hay una correlación entre estas dos variables. Esto es debido a que en matrices pequeñas no se llega a explotar la gran potencia paralela de la GPU. En cambio, cuanto más grande es la matriz más diferencia de rendimiento hay entre la versión serie contra la CUDA.

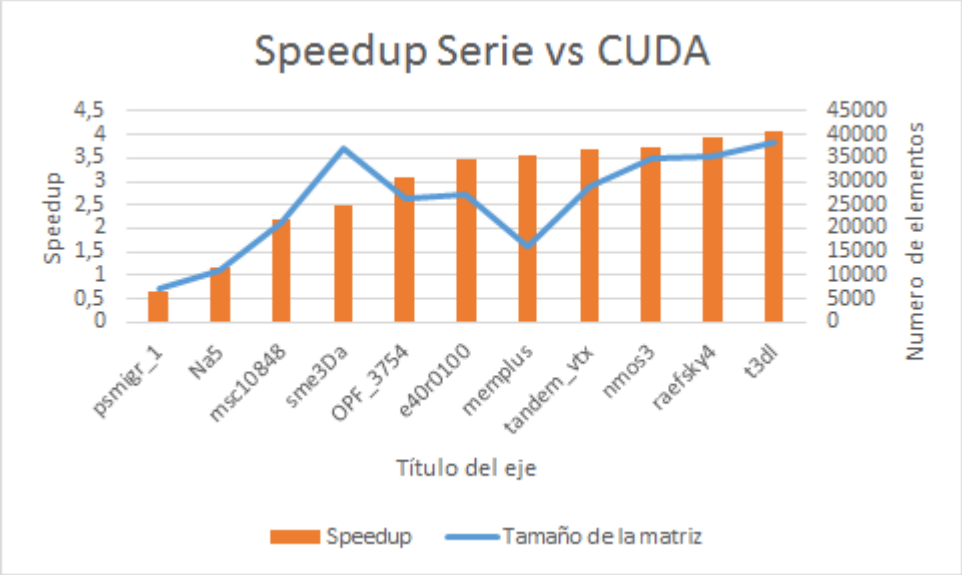


Figura 36: Speedup CUDA vs serie

11.1 Speedup cuSPARSE

La librería cuSPARSE[23] es una librería de NVIDIA que proporciona funciones para ejecutar operaciones sobre matrices dispersas. En ella está implementada la operación SpMV. En la librería se presenta los speedups que vemos en la figura 37. Los speedups conseguidos por la librería cuSPARSE y los obtenidos en este proyecto son bastante parecidos. En ambos casos se obtienen speedups entre 2 y 4 dependiendo de la matriz.

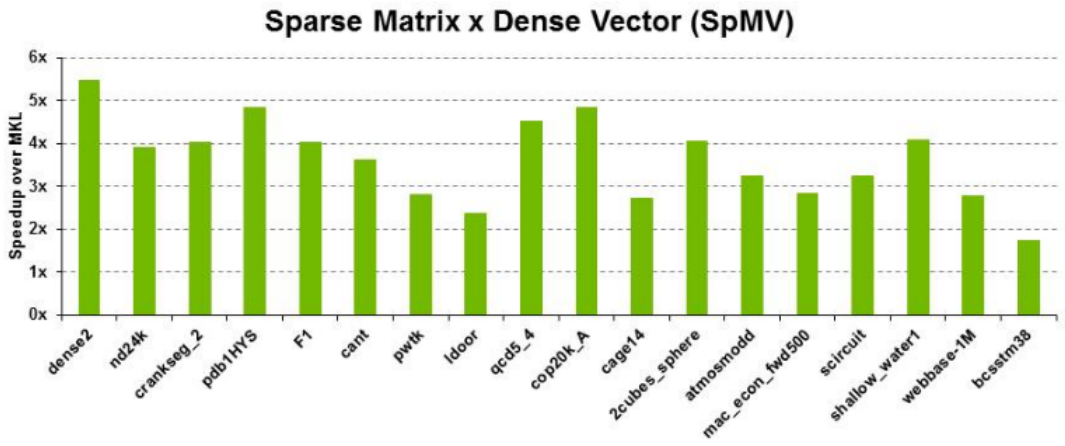


Figura 37: Speedup CUDA cuSPARSE

12 Conclusiones

Todos los formatos han demostrado ser más rápidos en sus versiones CUDA consiguiendo speedups de 2 a 3X. Demostrando la programación GPGPU ser un opción válida para la mejora del rendimiento de la operación SpMV.

Se ha encontrado que los formatos tienen un rendimiento altamente ligado a los patrones de las matrices. Por ello, los formatos que mejor rendimiento obtienen en media son los formatos con más flexibilidad. El formato CSR es el formato que consigue mejor rendimiento en serie y el formato HYB-G es el que mejor rendimiento consigue en CUDA.

En serie el formato CSR, siendo uno de los más simples, consigue un gran rendimiento debido a su independencia con los patrones de la matriz. Los accesos a memoria de este formato son totalmente regulares mitigando uno de los problemas raíz de la operación SpMV.

En CUDA el formato HYB-R logra el mejor rendimiento por varios motivos. El primero es que aporta una división del trabajo fácil y equitativa en bloques. El segundo es que dentro de cada bloque todos los threads deben realizar la misma cantidad exacta de trabajo. Por último, la mejora más importante es la distribución en memoria de la matriz. Esta distribución consigue que los accesos simultáneos en CUDA se hagan a posiciones cercanas en memoria. Esto maximiza el rendimiento de cada acceso que se hace a memoria.

Aunque estos dos formatos son los que han conseguido el mejor rendimiento en media, no son en todos los casos los mejores. Si la operación se debe ejecutar sobre matrices con patrones muy similares (patrones en banda, patrones en bloques) otros formatos que se adapten a sus particularidades como DIA o ELL puede superar en rendimiento a los formatos CSR y HYB-G.

Referencias

[1] *Biomecánica para predecir como se comporta la mandibula.* www.goo.gl/6kNShN

[2] *Método de los elementos finitos.* En Wikipedia. Recuerado el 2 de Enero de 2017 de www.goo.gl/xU4iJm

[3] Nvidia Compute Unified Device Architecture (CUDA). http://www.nvidia.com/object/cuda home new.html.

[4] Codigo ejemplo CUDA. http://www.nvidia.es/object/cuda-parallel-computing-es.html

[5] T. A. Davis and Y. Hu. The University of Florida. Colección de matrices dispersas. http://www.cise.ufl.edu/research/sparse/matrices/.

[6] Software de control de version GIT. https://git-scm.com/

[7] C. P., F. R., Fernández & Rodríguez, *Optimization of Sparse Matrix-Vector Multiplication Using Reordering Techniques on GPUs*, Galicia Supercomputing Center (CESGA), Santiago de Compostela, Spain 2011.

[8] E. Cuthill and J. McKee, & I. S, *Reducing the bandwidth of sparse symmetric matrices In Pro*, 24th Nat. Conf. ACM, pages 157–172, 1969.

[9] N. Bell, M. Garland, *Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors.* Conference on High Performance Computing Networking, Storage and Analysis.

[10] P. R., Amestoy, T. A. & Davis, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Analysis & Applic., Vol 17, no 4, pp. 886-905, Dec. 1996

[11] Juan C. Pichel, David E. Singh, and Jesús Carretero, *Reordering Algorithms for Increasing Locality on Multicore Processors*, Universidad Carlos III de Madrid, Spain, 2008.

[12] J. Willcock and A. Lumsdaine. *Accelerating sparse matrix computations via data compression. In Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 307–316, New York, NY, USA, 2006.

[13] K. Kourtis, G. Goumas, and N. Koziris. *Optimizing sparse matrix-vector multiplication using index and value compression.* In *Proceedings of the 5th Conference on Computing frontiers*, CF '08, pages 87–96, New York, NY, USA, 2008

[14] T. T., J. T., Ray, W. W., Chen , Kuo , S. M., J. T. & Wong, *Accelerating Sparse Matrix-Vector Multiplication on GPUs using Bit-Representation-Optimized Schemes*, Department of Computer Science, School of Computing, National University of Singapore, Singapore, 2013.

[15] D. Barbieri, V. Cardellini, S. Filippone, *A new approach for sparse matrix vector product on NVIDIA GPUs*. Universidad de Roma "Tor Vergata", Departamento de informática, Sistemas y Produccion,Technical Report RR-12.90 Enero 12, 2012

[16] F. Vazquez, J. J. Fernández, E. M. Garzón, *Sparse Computations on GPGPUs*. Practice and Experience 23 (8) (2011) 815–826.

[17] Marco Maggionia, Tanya Berger-Wolfa *An architecture-aware technique for optimizing sparse matrix-vector multiplication on GPUs*. International Conference on Computational Science, ICCS 2013.

[18] R. Grimes, D. Kincaid, and D. Young *ITPACK 2.0 User's Guide* Technical Report CNA-150, Center for Numerical Analysis, University of Texas, August 1979

[19] N. Bell and M. Garland, *Efficient sparse matrix-vector multiplication on CUDA*, NVIDIA Technical Report, NVR-2008-004, NVIDIA Corporation, 2008.

[20] Christian Felber, *"La economía del bien común"*, Deusto S.A. Ediciones. ISBN 9788423412808. 2011.

[21] Precio GFlop CPU http://info.grcooling.com/picking-the-right-processor

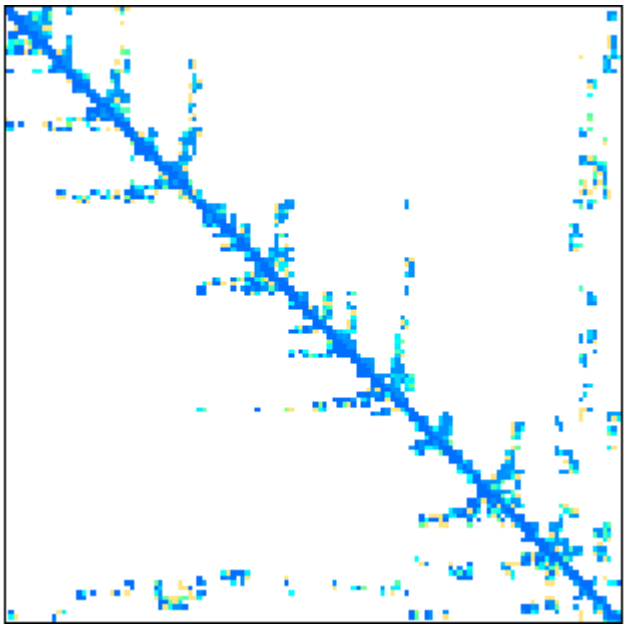
[22] Precio GFlop GPU http://aiimpacts.org/wikipedia-history-of-gflops-costs/

[23] cuSparse https://developer.nvidia.com/cusparse

13 Anexo matrices

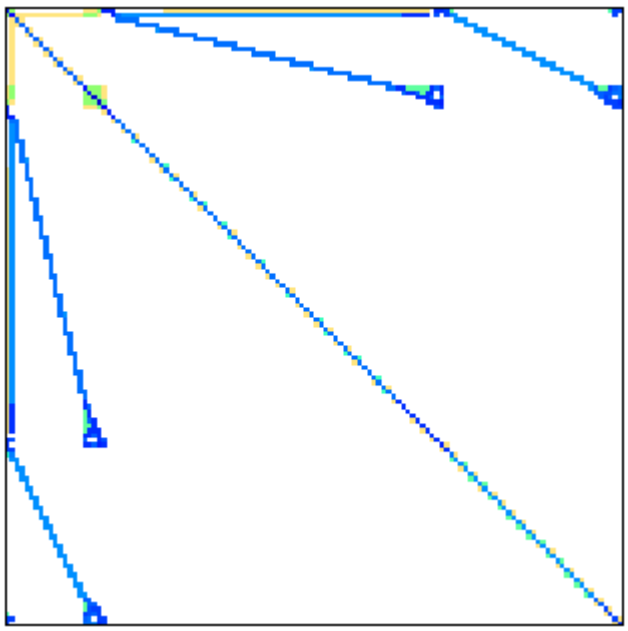
En este anexo se añaden las diferentes matrices usadas en las pruebas para un mejor entendimiento de los resultados.

13.1 e40r0100



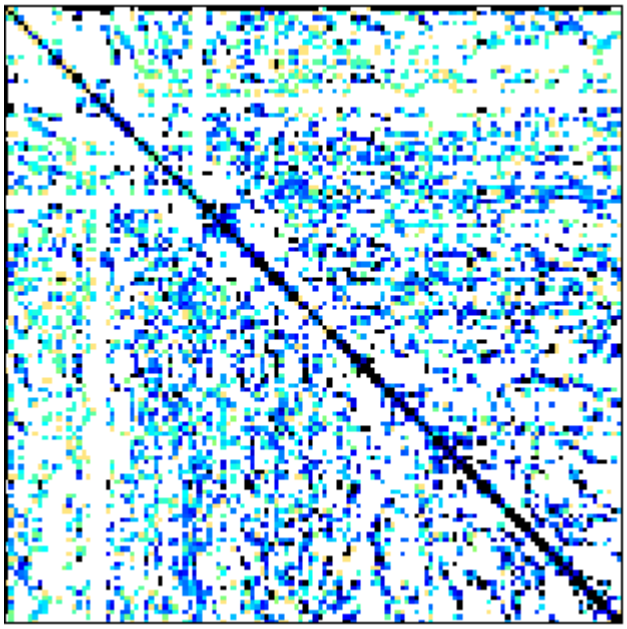
Tamaño	17,281x17,281
Nnz	553,562
%Nnz	0.18 %

13.2 memplus



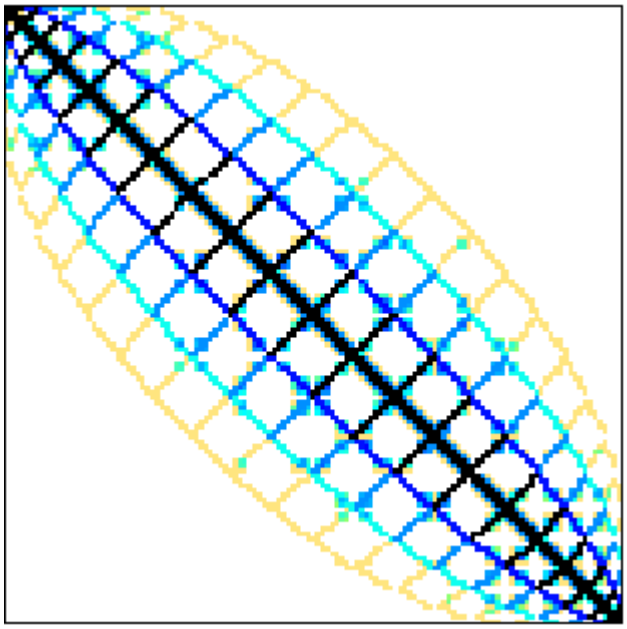
Tamaño	17,758x17,758
Nnz	99,147
%Nnz	0.04 %

13.3 msc10848



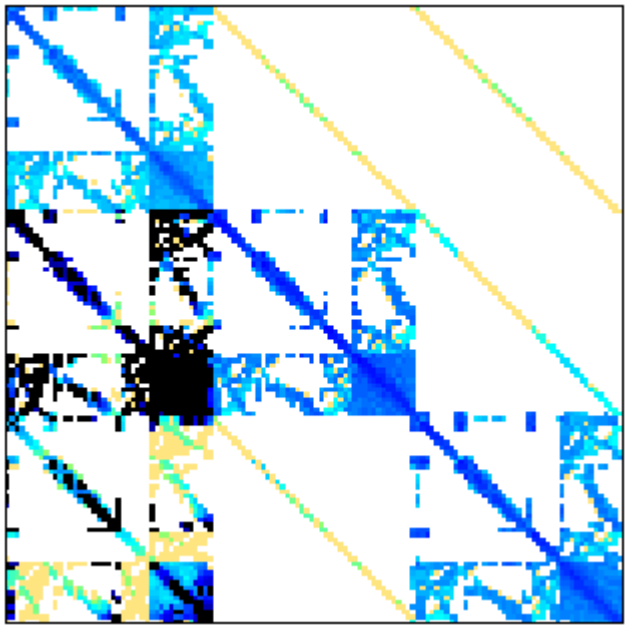
Tamaño	10,848x10,848
Nnz	1,229,776
%Nnz	1.04 %

13.4 Na5



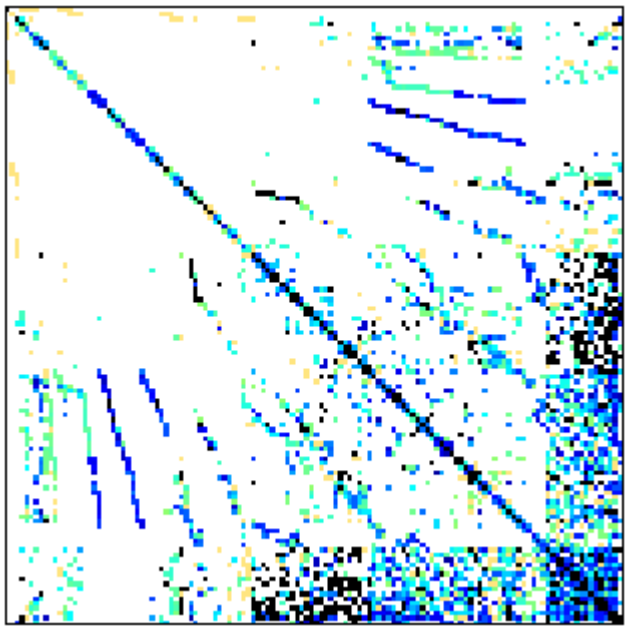
Tamaño	5,832x5,832
Nnz	305,630
%Nnz	0.9 %

13.5 nmos3



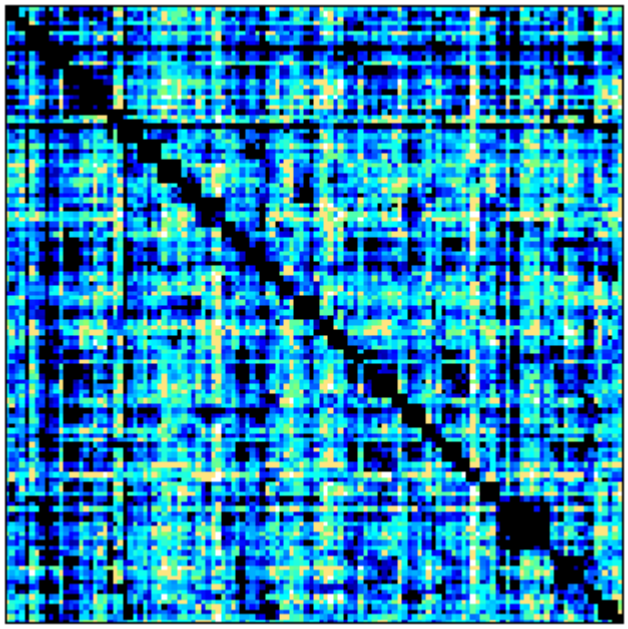
Tamaño	18,588x18,588
Nnz	237,130
%Nnz	0.11 %

13.6 OPF_3754



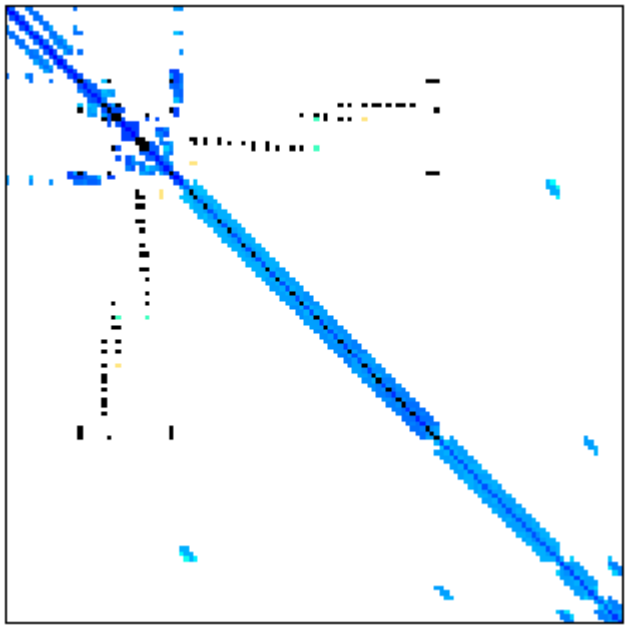
Tamaño	15,435x15,435
Nnz	141,478
%Nnz	0.05 %

13.7 psmigr_1



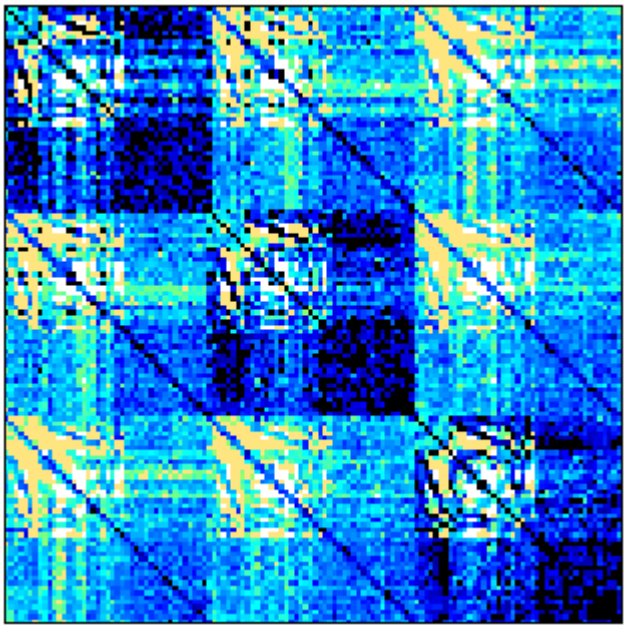
Tamaño	3,140x3,140
Nnz	543,160
%Nnz	5.50 %

13.8 raefsky4



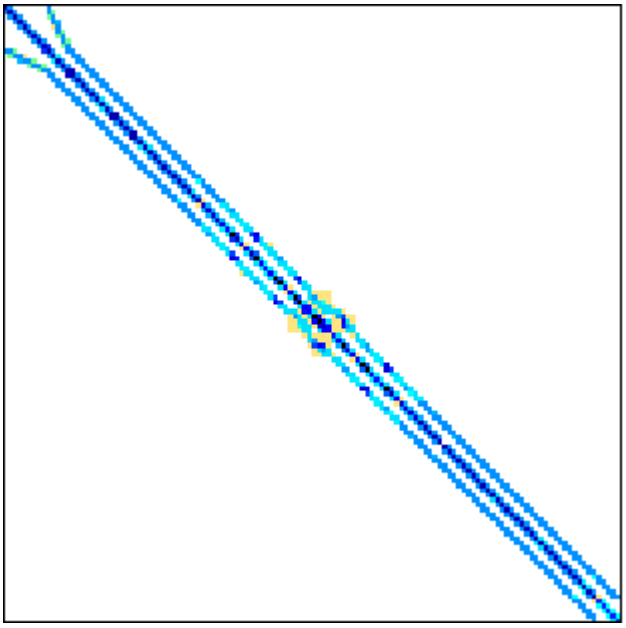
Tamaño	19,779x19,779
Nnz	1,316,789
%Nnz	0.34 %

13.9 sme3Da



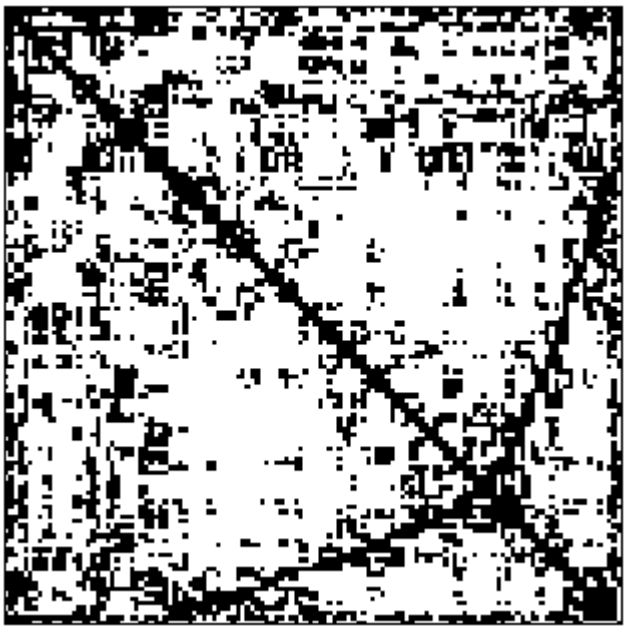
Tamaño	12,504x12,504
Nnz	874,887
%Nnz	0.56 %

13.10 t3dl_a



Tamaño	20,360x20,360
Nnz	509,866
%Nnz	0.12%

13.11 tandem_vtx



Tamaño	18,454x18,454
Nnz	253,350
%Nnz	0.07%

14 Anexo código

En este anexo primero se muestra un ejemplo de una llamada a un kernel de CUDA, en este caso para el formato ELL. Seguidamente se presenta cada formato siguiendo la siguiente estructura: algoritmo para pasar de formato denso al formato en cuestión, multiplicación en serie y multiplicación en CUDA.

14.1 Ejemplo llamada a kernel ELL

```
void MAIN_ELL_CUDA(){

   // Host variables
   float *A_ELL,
   int *Indices;

   // Calculate max elements in a row
   unsigned int Max;
   MaxNumberElementsRow(A,M,N,Max);

   // Get memory from host
   A_ELL = (float*) calloc(Max*M,sizeof(float));
   Indices = (int*) calloc(Max*M,sizeof(int));

   // Transform dense matrix to ELL matrix
   matrix_to_ELL(A,A_ELL,Indices,M,N,Max);

   // CUDA variables
   float *A_ELL_CUDA, *B_CUDA, *C_ELL_CUDA;
   int *Indices_CUDA;

   // Get memory from GPU
   cudaMalloc((float**)&A_ELL_CUDA, Max*M*sizeof(float));
   cudaMalloc((float**)&B_CUDA, N*sizeof(float));
   cudaMalloc((int**)&Indices_CUDA, Max*M*sizeof(int));
   cudaMalloc((float**)&C_ELL_CUDA, M*sizeof(float));

   // Copy matrix and vector to GPU
   cudaMemcpy(A_ELL_CUDA,  A_ELL, Max*M*sizeof(float) ,
       cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(B_CUDA, B, N*sizeof(float) , cudaMemcpyHostToDevice);
    cudaMemcpy(Indices_CUDA,Indices, Max*M*sizeof(int) ,
        cudaMemcpyHostToDevice);
    cudaMemcpy(C_ELL_CUDA, C_ELL, M*sizeof(float) ,
        cudaMemcpyHostToDevice);

    // Llamada al kernel de CUDA
    matrix_vector_CUDA_ELL<<<NUM_BLOCKS,BLOCK_SIZE>>>(A_ELL_CUDA,
        Indices_CUDA, B_CUDA, C_ELL_CUDA, M, Max);

    // Error control
    gpuErrchk( cudaPeekAtLastError() );
    gpuErrchk( cudaDeviceSynchronize() );

    // Copy the results from device to host
    cudaMemcpy(C_ELL, C_ELL_CUDA,M*sizeof(float),
        cudaMemcpyDeviceToHost);

    // Free CUDA
    cudaFree(A_ELL_CUDA);
    cudaFree(B_CUDA);
    cudaFree(Indices_CUDA);
    cudaFree(C_ELL_CUDA);

    // Free CPU
    free(A_ELL);
    free(Indices);
}
```

14.2 COO

```
void matrix_to_COO(float *A, int* Rows, int* Columns, float*
    Values, unsigned M, unsigned N, unsigned int &nnz ){
   int end = 0;
   for (int i=0; i<M; i++) {
      for (int j=0; j<N; j++){
         if(A[i*N+j] != 0){
            Rows[end] = i;
```

```
            Columns[end] = j;
            Values[end] = A[i*N+j];
            ++end;
         }
      }
   }
}

void matrix_vector_serial_COO(float *B, float *C, int *Rows, int
    *Columns, float *Values, unsigned int nnz){
   if (nnz != 0){
      for (int i=0; i<nnz; i++) {
         C[Rows[i]] += Values[i] * B[Columns[i]];
      }
   }
}

__global__ void matrix_vector_CUDA_COO_ATOMIC(float *B, float *C,
    int *Rows, int *Columns, unsigned int valuesPerThread, float
    *Values, unsigned int nnz){
   float tmp = 0;
   int id = blockIdx.x*blockDim.y+threadIdx.y;

   for (int i = id*valuesPerThread; i <
       (id*valuesPerThread)+valuesPerThread and i < nnz;++i){
      tmp = Values[i] * B[Columns[i]];
      atomicAdd(&C[Rows[i]],tmp);
   }

}
```

14.3 CSR

```
void matrix_to_CSR(float *A, int* Rows, int* Columns, float*
    Values, unsigned M, unsigned N, unsigned int &nnz ){
   int end = 0;
   int i_CSR = 1;
   Rows[0] = 0;
```

```
   for (int i=0; i<M; i++) {
      Rows[i_CSR] = Rows[i_CSR-1];
      for (int j=0; j<N; j++){
         if(A[i*N+j] != 0){
            Rows[i_CSR] += 1;
            Columns[end] = j;
            Values[end] = A[i*N+j];
            ++end;
         }
      }
      ++i_CSR;
   }
}

Void matrix_vector_serial_CSR(float *B, float *C, int *Rows, int
    *Columns, float *Values, unsigned int nnz, unsigned int M){
   if (nnz != 0){
      float tmp = 0;
      for (int i=0; i<M; i++) {
         for (int j=Rows[i]; j< Rows[i+1]; ++j) {
            tmp += Values[j] * B[Columns[j]];
         }
         C[i] = tmp;
         tmp = 0;
      }
   }
}


__global__ void matrix_vector_CUDA_CSR(float *B, float *C, int
    *Rows, int *Columns, float *Values, unsigned int nnz){

    int ROW = blockIdx.x*blockDim.y+threadIdx.y;
    int COL = blockIdx.x*blockDim.x+threadIdx.x;
   float tmp = 0;
   for (int j=Rows[ROW]; j< Rows[ROW+1]; ++j) {
      tmp += Values[j] * B[Columns[j]];
   }
   C[ROW] = tmp;
}
```

14.4 DIA

```
void matrix_to_DIA(float *A, float* A_DIA, int* Offset, unsigned
    int M, unsigned int N, unsigned int Max ){
   int i = M-1;
   int j = 0;
   int i_DIA = 0;
   while(i > 0){
      if (not isDiagonalEmpty(A, M, N, i, j)){
         for (int ii = i, jj = j ; ii < M and jj < N; ++ii, ++jj){
            A_DIA[Max*(jj-j)+i_DIA] = A[ii*N+jj];
         }
         Offset[i_DIA] = -1*i;
         ++i_DIA;
      }
      --i;
   }
   while (j < N){
      if (not isDiagonalEmpty(A, M, N, i, j)){
         for (int ii = i, jj = j ; ii < M and jj < N; ++ii, ++jj){
            A_DIA[Max*(jj-j)+i_DIA] = A[ii*N+jj];

         }
         Offset[i_DIA] = j;
         ++i_DIA;
      }
      ++j;
   }
}

void matrix_vector_serial_DIA(float *A_DIA, int *Offset, float *B,
    float *C, unsigned int M, unsigned int N){
   for (int i=0; i<N; i++) {
      for (int j=0; j<M; j++) {
         if (Offset[i] > 0) {
            C[j] += A_DIA[j*N+i] * B[Offset[i] + j];
         }
         else {
            C[j+Offset[i]*-1] += A_DIA[j*N+i] * B[j];
         }
```

```
      }
   }
}

__global__ void matrix_vector_CUDA_DIA(float *A_DIA, int *Offset,
    float *B, float *C, unsigned int M, unsigned int Num_diag){

    int ROW = blockIdx.x*blockDim.y+threadIdx.y;
    int i = ROW;
    float tmp;
   for (int j=0; j<Num_diag and ROW < M-Offset[j]; j++) {
         if (Offset[j] <= 0){
            tmp = A_DIA[Num_diag*ROW+j] * B[ROW];
            atomicAdd(&C[Offset[j]*-1+ROW], tmp);
         }
         else {
            tmp = A_DIA[Num_diag*ROW+j] * B[Offset[j]+ROW];
            atomicAdd(&C[ROW],tmp);

         }
      }
}
```

14.5 ELL

```
void matrix_to_ELL(float *A, float* A_ELL, int* Indices, unsigned
    int M, unsigned int N, unsigned int &Max ){
   int j_A_ELL;
   for (int i=0; i<M; i++) {
      j_A_ELL = 0;
      for (int j=0; j<N; j++){
         if(A[i*N+j] != 0){
            A_ELL[Max*i + j_A_ELL] = A[i*N+j];
            Indices[Max*i + j_A_ELL] = j;
            ++j_A_ELL;
         }
      }
   }
```

```
}

void matrix_vector_serial_ELL(float *A_ELL, int *Indices, float *B,
    float *C, unsigned int M, unsigned int N){
   for (int i=0; i<M; i++) {
      float tmp = 0.0;
      for (int j=0; j<N; j++)
         tmp += A_ELL[i*N+j] * B[Indices[i*N+j]];
      C[i] = tmp;
   }
}

__global__ void matrix_vector_CUDA_ELL(float *A_ELL, int *Indices,
    float *B, float *C, unsigned int M, unsigned int N){

    int ROW = blockIdx.x*blockDim.y+threadIdx.y;
    float tmpSum = 0;

    if (ROW < M) {
      for (int j=0; j<N; j++) {
         tmpSum += A_ELL[ROW*N+j] * B[Indices[ROW*N+j]];
        }
      C[ROW] = tmpSum;
    }
}
```

14.6 ELL-BASED

```
void calcularMaxOfSlice(float *A,unsigned int M, unsigned int N,
    unsigned int Slice, int ii, unsigned int &Max){

   Max = 0;
   int auxMax;
   for (int i = ii; i<ii+Slice and ii < M; i++) {
      auxMax = 0;
      for (int j=0; j<N; j++){
         if(A[i*N+j] != 0) ++auxMax;
      }
```

```
      if (auxMax > Max) Max = auxMax;
   }
}
void DENSE_to_ELL_BASED(float *A, float* &A_ELL_SLICE, int*
    Indices, int* Offset, int* K, unsigned int M, unsigned int N,
    unsigned int Slice){
   int Z = 0; //Index for vector K and Offset.
   int lastPosition = 0;

   for (int i=0; i<M; i += Slice) {
      unsigned int Max;
      calculateMaxOfSlice(A,M,N,Slice,i,Max);
      K[Z] = Max;

      for (int ii=0; ii<Slice and ii < M; ++ii) {
         int a = 0; //index for matrix A
         for (int j=0; j<N; j++){
            if(A[(i+ii)*N+j] != 0){
               A_ELL_SLICE[lastPosition+ Max*ii+a] = A[(i+ii)*N+j];
               Indices[lastPosition+ Max*ii+a] = j;
               ++a;
            }
         }
      }
      Offset[Z] = lastPosition;
      lastPosition = lastPosition + Slice * Max;
      ++Z;
   }
}

void matrix_vector_serial_ELL_BASED(float *A_ELL, int *Indices,
    int* Offset, int* K, float *B, float *C, unsigned int M,
    unsigned int N, unsigned int Slice){
   for (int i=0; i<(M/Slice); i++) {
      N = K[i];
      for (int s = 0; s < Slice; ++s){
         float tmp = 0.0;
         for (int j=0; j<N; j++) {
            tmp += A_ELL[Offset[i] + s*N +j] * B[Indices[Offset[i] +
                s*N +j]];
```

```
        }
        C[i*Slice+s] = tmp;
      }
   }
}

__global__ void matrix_vector_CUDA_ELL_BASED(float *A_ELL, int
    *Indices, int* Offset, int* K, float *B, float *C, unsigned int
    M, unsigned int Slice){

   int ROW = blockIdx.x*blockDim.y+threadIdx.y;
   if (ROW < M) {
      int i = ROW/Slice;
      int N = K[i];
      int s = ROW % Slice;

      float tmp = 0.0;

      for (int j=0; j<N; j++) {
         tmp += A_ELL[Offset[i] + s*N + j] * B[Indices[Offset[i] +
             s*N + j]];
      }
      C[ROW] = tmp;
   }
}
```

14.7 ELL-G

```
void matrix_ELL_to_ELL_G(float *A_ELL, float* A_ELL_Col, int*
    Indices, int* Indices_Col, unsigned int M, unsigned int N,
    unsigned int &Max ){
   int end = 0;
   for (int i=0; i<Max;i++) {
      for (int j=i; j<(M*Max); j=j+Max){
            Indices_Col[end] = Indices[j];
            A_ELL_Col[end] = A_ELL[j];
            ++end;
      }
```

```
   }
}

void matrix_vector_serial_ELL_G(float *A_ELL_col, int *Indices_col,
    float *B, float *C, unsigned int M, unsigned int N){
   for (int i=0; i<M; i++) {
      float tmp = 0.0;
      for (int j=i; j<N*M; j= j+M)
         tmp += A_ELL_col[j] * B[Indices_col[j]];
      C[i] = tmp;
   }
}

__global__ void matrix_vector_CUDA_ELL_G(float *A_ELL_col, int
    *Indices_col, float *B, float *C, unsigned int M, unsigned int
    N){
    int ROW = blockIdx.x*blockDim.y+threadIdx.y;
    float tmpSum = 0;

    if (ROW < M) {
      for (int j=ROW; j<N*M; j= j+M){
         tmpSum += A_ELL_col[j] * B[Indices_col[j]];
        }
    }
    C[ROW] = tmpSum;

}
```

14.8 HYB

```
void matrix_to_HYB(float *A,int* Rows, int* Columns, float* Values,
    float* A_ELL, int* Indices, unsigned int M, unsigned int N,
    unsigned int Max){
   int j_A_ELL;
   int end = 0;
   for (int i=0; i<M; i++) {
      j_A_ELL = 0;
```

```
      for (int j=0; j<N; j++){
         if(A[i*N+j] != 0){
            if (j_A_ELL < Max){
               A_ELL[Max*i+j_A_ELL] = A[i*N+j];
               Indices[Max*i+j_A_ELL] = j;
               ++j_A_ELL;
            }
            else{
               Rows[end] = i;
               Columns[end] = j;
               Values[end] = A[i*N+j];
               ++end;
            }
         }
      }
   }
}


void matrix_vector_serial_COO_HYB(float *B, float *C, int *Rows,
    int *Columns, float *Values, unsigned int nnz){
   if (nnz != 0){
      for (int i=0; i<nnz; i++) {
         C[Rows[i]] += Values[i] * B[Columns[i]];
      }
   }
}
void matrix_vector_serial_ELL_HYB(float *A_ELL, int *Indices, float
    *B, float *C, unsigned int M, unsigned int N){
   for (int i=0; i<M; i++) {
      float tmp = 0.0;
      for (int j=0; j<N; j++)
         tmp += A_ELL[i*N+j] * B[Indices[i*N+j]];
      C[i] = tmp;
   }
}
__global__ void matrix_vector_CUDA_COO_HYB_ATOMIC(float *B, float
    *C, int *Rows, int *Columns, unsigned int valuesPerThread, float
    *Values, unsigned int nnz){
   float tmp = 0;
```

```
   int id = blockIdx.x*blockDim.y+threadIdx.y;

   for (int i = id*valuesPerThread; i <
       (id*valuesPerThread)+valuesPerThread and i < nnz;++i){
      tmp = Values[i] * B[Columns[i]];
      atomicAdd(&C[Rows[i]],tmp);
   }
}

__global__ void matrix_vector_CUDA_ELL_HYB(float *A_ELL, int
    *Indices, float *B, float *C, unsigned int M, unsigned int N){

    int ROW = blockIdx.x*blockDim.y+threadIdx.y;
    float tmpSum = 0;

    if (ROW < M) {
      for (int j=0; j<N; j++) {
         tmpSum += A_ELL[ROW*N+j] * B[Indices[ROW*N+j]];
        }
      C[ROW] = tmpSum;
    }
}
____global__ void reduction(float *A, float *B, unsigned int N,
    unsigned int M){
   int id = blockIdx.x*blockDim.y+threadIdx.y;
   float tmp = 0.0;
   for (int i=0; i<M; ++i){
      tmp += A[i*M+id];
   }
   B[id] = tmp;
}
```

14.9 HYB-G

```
void matrix_to_HYB_G(float *A,int* Rows, int* Columns, float*
    Values, float* A_ELL, int* Indices, unsigned int M, unsigned int
    N, unsigned int Max){
```

```
   int j_A_ELL;
   int end = 0;
   for (int i=0; i<M; i++) {
      j_A_ELL = 0;
      for (int j=0; j<N; j++){
         if(A[i*N+j] != 0){
            if (j_A_ELL < Max){
               A_ELL[Max*i+j_A_ELL] = A[i*N+j];
               Indices[Max*i+j_A_ELL] = j;
               ++j_A_ELL;
            }
            else{
               Rows[end] = i;
               Columns[end] = j;
               Values[end] = A[i*N+j];
               ++end;
            }
         }
      }
   }
}


void matrix_vector_serial_COO_HYB_G(float *B, float *C, int *Rows,
    int *Columns, float *Values, unsigned int nnz){
   if (nnz != 0){
      for (int i=0; i<nnz; i++) {
         C[Rows[i]] += Values[i] * B[Columns[i]];
      }
   }
}
void matrix_vector_serial_ELL_HYB_G(float *A_ELL_col, int
    *Indices_col, float *B, float *C, unsigned int M, unsigned int
    N){
   for (int i=0; i<M; i++) {
      float tmp = 0.0;
      for (int j=i; j<N*M; j= j+M)
         tmp += A_ELL_col[j] * B[Indices_col[j]];
      C[i] = tmp;
   }
```

```
}
__global__ void matrix_vector_CUDA_COO_HYB_G_ATOMIC(float *B, float
    *C, int *Rows, int *Columns, unsigned int valuesPerThread, float
    *Values, unsigned int nnz){
   float tmp = 0;
   int id = blockIdx.x*blockDim.y+threadIdx.y;

   for (int i = id*valuesPerThread; i <
       (id*valuesPerThread)+valuesPerThread and i < nnz;++i){
      tmp = Values[i] * B[Columns[i]];
      atomicAdd(&C[Rows[i]],tmp);
   }
}

__global__ void matrix_vector_CUDA_ELL_HYB_G(float *A_ELL_col, int
    *Indices_col, float *B, float *C, unsigned int M, unsigned int
    N){
    int ROW = blockIdx.x*blockDim.y+threadIdx.y;
    float tmpSum = 0;

    if (ROW < M) {
      for (int j=ROW; j<N*M; j= j+M){
         tmpSum += A_ELL_col[j] * B[Indices_col[j]];
        }
    }
    C[ROW] = tmpSum;

}
__global__ void reduction(float *A, float *B, unsigned int N,
    unsigned int M){
   int id = blockIdx.x*blockDim.y+threadIdx.y;
   float tmp = 0.0;
   for (int i=0; i<M; ++i){
      tmp += A[i*M+id];
   }
   B[id] = tmp;
}
```